LNCS 3463

Mario Dal Cin
Mohamed Kaâniche
András Pataricza (Eds.)

Dependable Computing – EDCC-5

5th European Dependable Computing Conference
Budapest, Hungary, April 2005
Proceedings

Springer

Mario Dal Cin   Mohamed Kaâniche
András Pataricza (Eds.)

Dependable Computing – EDCC-5

5th European Dependable Computing Conference
Budapest, Hungary, April 20-22, 2005
Proceedings

Springer

Volume Editors

Mario Dal Cin
University of Erlangen-Nürnberg, Institute for Computer Sciences III
Martensstr. 3, 91058 Erlangen, Germany
E-mail: dalcin@informatik.uni-erlangen.de

Mohamed Kaâniche
LAAS-CNRS
7, Avenue du Colonel Roche, 31077 Toulouse Cedex 04, France
E-mail: mohamed.kaaniche@laas.fr

András Pataricza
Budapest University of Technology and Economics
Department of Measurement and Instrument Engineering
Magyar tudósok körútja 2. B. 420, 1502 Budapest, Hungary
E-mail: pataric@mit.bme.hu

Library of Congress Control Number: 2005924257

CR Subject Classification (1998): B.1.3, B.2.3, B.3.4, B.4.5, C.3-4, D.2.4, D.2.8, C.2

ISSN 0302-9743
ISBN-10 3-540-25723-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-25723-3 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11408901 06/3142 5 4 3 2 1 0

Foreword

It is always a special honor to chair the European Dependable Computing Conference (EDCC). EDCC has become one of the well-established conferences in the field of dependability in the European research area. Budapest was selected as the host of this conference due to its traditions in organizing international scientific events and its traditional role of serving as a meeting point between East and West.

EDCC-5 was the fifth in the series of these high-quality scientific conferences. In addition to the overall significance of such a pan-European event, this year's conference was a special one due to historic reasons. The roots of EDCC date back to the moment when the Iron Curtain fell. Originally, two groups of scientists from different European countries in Western and Eastern Europe – who were active in research and education related to dependability – created a joint forum in order to merge their communities as early as in 1989. This trend has continued up to today. This year's conference was the first one where the overwhelming majority of the research groups belong to the family of European nations united in the European Union. During the past 16 years we observed that the same roots in all the professional, cultural and scientific senses led to a seamless integration of these research communities previously separated artificially for a long time.

EDCC has become one of the main European platforms to exchange new research ideas in the field of dependability. Its pillars are three national groups: the SEE Working Group "Dependable Computing" in France, the GI/ITG/GMA Technical Committee on Dependability and Fault Tolerance in Germany, and the AICA Working Group on Dependability of Computer Systems in Italy. Additionally, several professional international organizations of worldwide scope, like IEEE and IFIP, graciously supported this conference from the very beginning.

Obviously, a conference has to follow the development of its focus area. This year, the trend to incorporate new topics was observable in the number of submissions and in the variety of topics. At the same time, a main objective of the organizers was to strengthen the pan-European nature of this scientific event according to the tradition of EDCC. An additional objective was – as defined in the guidelines of the Steering Committee – to attract more young people to contribute to the conference not only through the presentation of their research work but also through organizing important events. This objective was served by inviting new members to the Program Committee who had already proven their scientific abilities but previously did not take an active part in organizing EDCC.

In addition to the traditional forms of presentations such as fast abstracts and invited speakers presenting the state of the art in industrial practice, new forums were introduced to this year's conference. Special attention was paid to educational aspects. A panel was organized to identify future challenges in research and harmonize the evolution of dependability-related education with the ongoing Bologna process. A special track for student papers was organized in order to support young researchers in contributing to one of the major international conferences. Representative projects from all over Europe were invited to present their work in progress and to provide a forum for discussing their results and help the dissemination of the main achievements.

The large number of contributions necessitated that, for the first time in the EDCC series, EDCC-5 was held in two tracks. We hope that the impact of this splitting will be that everybody in the audience will find a topic that best fits his or her personal professional interests.

The preparation of such an important international conference requires always enthusiastic support from supporting teams. We had the full support offered by excellent teams. We hope that the conference chairs helped not only to share the wide spectrum of responsibility but also to reach a synergetic effect between their particular fields. The organizing teams including their Chairs, Wolfgang Hohl, Tamás Bartha and Dániel Varró, and numerous members were extremely helpful; their work contributed to the success of the conference. The website was maintained by László Gönczy. Their enthusiastic work is acknowledged.

Our special thanks goes to Mohamed Kaâniche, the Program Chair, for his thoughtful planning of the Program Committee Meeting. His efforts were instrumental in setting up an excellent scientific program. A large number of referees helped the Program Committee in evaluating the papers and special thanks should be dedicated to all of them, not only for carefully judging the quality of the papers but also for providing the authors with feedback which will hopefully help in their future work as well.

We would also like to thank the Education and Perspectives Chair, Henrique Madeira, and the Student Forum Chair, Miroslaw Malek, for their work.

The host institutions of the Chairs offered valuable help for the conference. LAAS-CNRS helped us to organize the Program Committee Meeting held in Toulouse; the Friedrich-Alexander-Universität in Erlangen and the Budapest University of Technology and Economics helped us in every imaginable way. Without their general support the conference would not have been as successful as we hoped it would be. The Budapest University of Technology and Economics, as host institution, was supported by Öt Évszak Ltd., a conference organizing enterprise, in all the preparations of the conference, including the management of finances.

For us to organize this conference was a challenge and we hope that the participants were satisfied with both the scientific quality and the organization of the conference.

February 2005

Mario Dal Cin
András Pataricza

Preface

On behalf of the Program Committee of the fifth edition of the European Dependable Computing Conference (EDCC-5), it was my pleasure to welcome attendees to Budapest, Hungary.

EDCC aims to provide a European venue for researchers and practitioners from all over the world to present and discuss their latest research results and developments. The conference aims to cover the many dimensions of dependability and fault tolerance, encompassing fundamental theoretical approaches, practical experimental projects, and commercial components and systems.

In recent years, the importance of dependability has been increasing beyond the classical critical application domains (telecommunications, transportation, space, etc.) as many other domains of the economy (commerce, finance, energy distribution) are recognizing the dependability of worldwide information and communication infrastructures to be one of their top problems. Besides traditional hardware and software faults, concerns include human interaction faults, they being accidental or malicious. The contributions received this year aim at addressing these problems and challenges, and cover various dimensions of dependable computing, including architecture design, protocols, verification, measurement and evaluation.

I express my thanks to the 33 Program Committee members for their hard work and continual involvement, starting with their assistance in advertising the conference. This resulted in the second largest number of submissions, from a broad range of institutions from academia and industry, since the conference was established. Overall, we received 90 submissions, originating from 27 countries; 58 submissions were from Europe, 19 from USA and North and South America, and the other submissions were from Asia.

All the submissions were thoroughly reviewed and discussed by the PC members with the support of 147 external reviewers. The entire process was handled electronically. This greatly helped in reducing delays while allowing good information exchange among the reviewers and the Program Committee. A total of 317 reviews were received and all papers were reviewed by at least 3 referees. I am very thankful to all the reviewers for their help and valuable inputs.

The Program Committee met in Toulouse on December 16–17, 2004 at LAAS-CNRS to select those papers judged to be of sufficiently high standard to be presented at the conference and to be included in the proceedings. Among the 90 submissions, we selected 30 contributions, corresponding to 21 regular papers, 5 practical experience reports and 4 prototype description tools. The selected papers cover many different areas of dependable computing, including fault tolerance, verification, analysis and evaluation, in both hardware and software.

In addition to regular sessions organized into two parallel tracks, the conference offered various opportunities for interaction and discussion through the organization of a Student Forum, Fast Abstracts sessions, and an invited session for presenting ongoing dependability-related projects. Two keynote addresses and a panel on dependability challenges and education perspectives complemented the technical program.

EDCC-5 would not have been possible without support and participation from numerous individuals and organizations. I would like to thank all authors who submitted their work to the conference, the EDCC-5 Program Committee members, and the external reviewers for their outstanding help, support and effort in elaborating this year's program. I would like to express my appreciation to the Steering Committee and the Organizing Committee members whose support was essential for making EDCC-5 a success.

Special thanks go to Mario Dal Cin and András Pataricza, the General Chairs, and to Luca Simoncini, the Steering Committee Chair, for their constructive contributions and helpful advice. Also, I want to explicitly thank Tamás Bartha, Wolfgang Hohl, Henrique Madeira, Miroslaw Malek for their dedication in handling the Fast Abstracts, the Proceedings, the Panel and the Student Forum. Finally, I would like to express my gratitude to several people at LAAS-CNRS for their efficient help and support: Jean Arlat, Yves Crouzet, Karama Kanoun, Jean-Claude Laprie, Joëlle Penavayre and Jean-Michel Pons.

I hope that the participants benefited from the conference, and that the exchange of information that took place will help the dependable computing community to advance the engineering practice, research and standards.

Toulouse, February 2005

Mohamed Kaâniche
Program Chair

Organization

General Co-chairs

Mario Dal Cin
Univ. of Erlangen-Nürnberg
Germany

András Pataricza
Budapest Univ. of Technology
and Economics, Hungary

Program Chair

Mohamed Kaâniche
LAAS-CNRS, France

Publication Chair

Wolfgang Hohl
Univ. of Erlangen-Nürnberg, Germany

Education and Perspectives Chair

Henrique Madeira
Univ. of Coimbra, Portugal

Student Forum Chair

Miroslaw Malek
Humboldt Univ. Berlin, Germany

Fast Abstracts Chair

Tamás Bartha
Hungarian Academy of Sciences, Hungary

Local Arrangements Chair

Dániel Varró
Budapest Univ. of Technology and Economics, Hungary

International Liaison Chairs

North America:	William Sanders, UIUC, USA
South America:	Eliane Martins, Unicamp, Brazil
Asia:	Takashi Nanya, Univ. of Tokyo, Japan

EDCC Steering Committee

Chair: Luca Simoncini, Italy

Algirdas Avižienis, Lithuania
Mario Dal Cin, Germany
Karl-Erwin Großpietsch, Germany
Karama Kanoun, France
Johan Karlsson, Sweden
Hermann Kopetz, Austria
Jean-Claude Laprie, France
András Pataricza, Hungary
Brian Randell, UK
João Gabriel Silva, Portugal
Janusz Sosnowski, Poland
Raimund Ubar, Estonia

Program Committee

Gustavo Alonso
Roberto Baldoni
Iain Bate
Peter Bishop
Jean-Paul Blanquart
Andrea Bondavalli
Diamantino Costa
Sadie Creese
Marc Dacier
Geert Deconinck
Felicita Di Giandomenico
Susanna Donatelli
Christof Fetzer
Gerhard Fohler
Elena Gramatova
Rachid Guerraoui
Zoltán Hornák
Ricardo Jimenez-Peris
Jan Jürjens
Johan Karlsson
Christian Landrault
István Majzik
Edgar Nett
Ondrej Novak
Stanislaw Piestrak
Jaan Raik
Manfred Reitenspieß
Alexander Romanovsky
Gerardo Rubino
José Rufino
Juan-Carlos Ruiz
Elena Troubitsyna
Hélène Waeselynck

External Referees

Agbaria, Adnan
Almeida, Carlos
Alvisi, Lorenzo
Anceaume, Emmanuelle
Andrews, Anneliese A.
Arief, L.B.
Arlat, Jean
Avizienis, Algirdas
Bade, R.
Bagchi, Saurabh
Bernardeschi, Cinzia
Bernardi, Simona
Bertolino, Antonia
Bieber, Pierre
Blanc Clavero, Sara
Bloomfield, Robin
Bobbio, Andrea
Botaschajan, Jewgenij
Brooke, Phil
Cachin, Christian
Cai, Xia
Cancela, Héctor
Chiaradonna, Silvano
Coleman, J.W.
Correia, Miguel
Coutinho, Manuel
Crispo, Bruno
Cukier, Michel
Cuppens, Frederic
de Andrés Martínez, D.
Defago, Xavier
Delporte-Gallet, Carole
Deswarte, Yves
Dohi, Tadashi
Dondossola, Giovanna
Elnozahy, E.N.
Eugster, Patrick
Ezhilchelvan, Paul
Fabre, Jean-Charles
Fauconnier, Hugues
Felber, Pascal
Felici, Massimo
France, Robert B.
Fristacky, Norbert
Frühauf, Karol
Garcia, A.F.
Garg, Sachin
Gaudel, Marie-Claude
Geier, Martin
Georg, Geri
Gil Vicente, Pedro
Gossens, Stefan
Gradinariu, Maria
Grandoni, Fabrizio
Greve, Fabíola G.P.
Großpietsch, K.-E.
Gyimóthy, Tibor
Haverkort, Boudewijn R.
Herms, Andre
Hohl, Wolfgang
Horvath, Andras
Hurfin, Michel
Istrate, Dan
Jimenez-Merino, Ernesto
Kanoun, Karama
Keidar, Idit
Kemme, Bettina
Kharchenko, V.
Kikuno, Tohru
Killijian, Marc-Olivier
Kof, Leonid
Kolar, Milan
Koopman, Philip
Kuball, Silke
Kuo, Sy Yen
Kursawe, Klaus
Labiche, Yvan
Laibinis, Linas
Laprie, Jean-Claude
Larrea, Mikel
Le Traon, Yves
Legall, Pascale
Littlewood, Bev
Lyu, Michael R.
Madeira, Henrique
Maehle, Erik
Maia, Ricardo
Malek, Miroslaw
Martinelli, Fabio
Martins, Eliane
Maxino, Theresa
Melhem, Rami
Milani, Alessia
Moraes, Regina Lucia O.
Morgan, Graham
Moustafoui, Achour
Nadjm-Tehrani, Simin
Neves, Nuno Ferreira
Ortalo, Rodolphe
Patino-Martinez, Marta
Pfitzmann, Andreas
Pister, Markus
Powell, David
Preneel, Bart
Puliafito, Antonio
Quisquater, J.-J.
Ramzan, Zulfikar
Randell, Brian
Rodriguez, Manuel
Roy, Matthieu
Ryan, Peter
Scarpa, M.
Schemmer, Stefan
Schiper, André
Schnieder, Eckehard
Sereno, Matteo
Silva, João Gabriel
Silva, Nuno
Simoncini, Luca
Simonot, Françoise
Sobe, Peter
Sonza Reorda, Matteo
Sosnowski, Janusz
Strigini, Lorenzo
Süsskraut, Martin
Szanto, Judith
Tai, Ann
Telek, Miklos
Tixeuil, Sebastien

Trikaliotis, Spiro	Vaidyanathan, K.	Xu, Zhongfu
Tsai, Jichiang	van Moorsel, Aad	Yuste, Pedro
Tschaeche, Oliver	Veríssimo, Paulo	Zarras, A.
Tucci Piergiovanni, Sara	Virgillito, Antonio	Zhu, Ji
Ubar, Raimund	Wagner, Stefan	
Urbán, Péter	Widder, Josef	

Table of Contents

A Process Toward Total Dependability – Airbus Fly-by-Wire Paradigm

Pascal Traverse, Isabelle Lacaze, and Jean Souyris

AIRBUS, 316 route de Bayonne, 31060 Toulouse cedex 03, France
{Pascal.Traverse, Isabelle.Lacaze, Jean.Souyris}@airbus.com

The presentation deals with digital electrical flight control system of the Airbus airplanes. This system is built to very stringent dependability requirements both in term of safety (the system must not output erroneous signals) and availability. System safety and availability principles along with assessment process are presented with an emphasis on their evolution and on future challenges.

The purpose of a flight control system is to link the pilot to the control surfaces of the airplane. A fly-by-wire system replaces mechanical transmission of signal to the actuators by a set of computers and electrical wires. Main impairments to such a system are erroneous positioning of control surfaces (safety) and loss of control (system availability).

With respect to physical faults, safety is ensured basically by the use of command and monitoring computers, such that in case of failure, the outputs are forced in a safe state. Redundancy provides the needed availability.

With respect to design and manufacturing errors, error avoidance and removal are applied with a stringent development process. Error tolerance is used as well.

Particular risks are a concern in the sense that they can be single events that could affect several redundancies. Segregation between redundant elements when they are installed is a key precaution.

Airbus flight control system offers piloting aids such as flight envelope protections, some of them are available on non fly-by-wire aircraft while others are specific, along with maintainability helping devices.

Safety assessment process allows for both qualitative and quantitative assessment, proceedings from aircraft top-level events.

Reference

1. Traverse, P., Lacaze, I., Souyris, J.: Airbus fly-by-wire: a total approach to dependability. 18th IFIP World Computer Congress – Topical session "fault tolerance for trustworthy and dependable information infrastructure" (Toulouse, France), Kluwer Academic Press, 2004, pp.191-212.

Building and Using Quorums Despite Any Number of Process of Crashes

Roy Friedman[1], Achour Mostefaoui[2], and Michel Raynal[2]

[1] Computer Science Department, Technion, Haifa 32000, Israel
roy@cs.technion.ac.il
[2] IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes, France
{achour , raynal}@irisa.fr

Abstract. Failure detectors of the class denoted $\mathcal{P}^t$ eventually suspect all crashed processes in a permanent way (completeness) and ensure that, at any time, no more than $n - t - 1$ alive processes are falsely suspected (accuracy), n being the total number of processes. This paper rst shows that a simple combination of such a failure detector with a two-step communication pattern can provide the processes with an interesting intersection property on sets of values. As an example illustrating the bene t and the property that such a combination can provide when designing protocols, a leader-based consensus protocol whose design relies on its systematic use is presented. Then the paper presents a $\mathcal{P}^t$-based protocol that builds quorums in systems where up to t processes can crash with $t < n$.

Keywords: Asynchronous system, Consensus, Distributed algorithm, Fault tolerance, Leader oracle, Quorum, Process crash, Unreliable failure detector.

1 Introduction

Context of the study. Quorums have been used for a long time in distributed systems [23]. A quorum is a set of processes satisfying some minimal requirements. A classical quorum example is the set of processes that have the last copy of a le; another example of quorum is the set of processes from which a process has received messages. To be useful, quorums are usually de ned as intersecting sets. When it is satis ed, this intersection property allows ensuring consistency requirements. One of the most known examples is the sets of read/write quorums used to ensure that any read operation always gets the last value of a replicated data. Quorums have also been used in communication protocols to ensure dissemination of data with a low message cost [3, 17]. Instead of broadcasting data from all to all (that would require n^2 messages where n is the total number of processes), each process sends its data only to a quorum of processes that aggregate the received data to compose a single (bigger) message, forward it to another quorum, etc., until all processes get a copy of each data. In this example, the tradeoff relating the number of forwarding steps (d) and the total number of messages used to implement such a communication pattern is investigated in [3] (where it is shown that "regular" systems allow a pattern of d communication steps with a $O(nd\sqrt[d]{n})$ message complexity, where n is the total number of processes).

In addition to data consistency and data dissemination, quorums have also been used (in conjunction with appropriate mechanisms) to solve agreement problems in asynchronous distributed systems prone to process crashes. The two most studied such problems are *consensus* and *non-blocking atomic commit*. Namely, in consensus, each process proposes a value, and the non-faulty processes have to decide a value (termination), such that no two different values are decided (uniform agreement), and the decided value is a proposed value (validity). The non-blocking atomic commit problem is de ned similarly, except for the validity property that is more involved. Namely, a process initially votes (i.e., it proposes the value yes or no), and the set of decided values is restricted to *commit* and *abort*. The validity requirements is then the following: *commit* can be decided only if all the processes have voted yes; however, even in such cases, if at least one process crashes, it is then allowed to decide *abort*.

As these problems have no solution in pure asynchronous systems [11], these systems have to be appropriately enriched in order to solve consensus or non-blocking atomic commit. A very attractive approach to enrich an asynchronous system is the *failure detector* approach [5]. A failure detector class is de ned by two properties: a *completeness* property that is on the actual detection of crashes, and an *accuracy* property that restricts the erroneous suspicions that a failure detector can make. For example, the failure detector class denoted $\Diamond\mathcal{S}$ includes all the failure detectors that (1) eventually suspect all crashed processes (strong completeness), and (2) ensure that eventually there is a correct process that is no longer suspected (eventual weak accuracy). This class is particularly interesting for the following reason. Despite the fact that the properties that de ne it are very weak (namely, except for one correct process that is no longer suspected after some unknown but nite time, the other correct processes can be arbitrarily suspected), it is the weakest class of failure detectors that allows to solve the consensus problem in asynchronous systems prone to up to $t < n/2$ process crashes [4].

The methodological construction of the $\Diamond\mathcal{S}$-based consensus protocol described in [20] shows explicitly how quorums and $\Diamond\mathcal{S}$ can be combined to solve consensus: $\Diamond\mathcal{S}$ is used to provide the consensus termination property, while majority quorums (that, due to the majority of correct processes assumption, can effectively be built) are used to ensure that the uniform agreement property is never violated. For the non-blocking atomic commit problem, it was shown that $?\mathcal{P}+\Diamond\mathcal{S}$ is the weakest class of timeless[1] failure detectors that allow it to be solved when a majority of processes do not crash [14], where $?\mathcal{P}$ is a failure detector that returns [illegible] when at least one process has already crashed, and [illegible] otherwise [13]. A $\Diamond S$-based solution to a somewhat weaker version of the atomic commit problem that relies on quorums and a majority of correct processes has been presented in [16] (extending the more limited solution described in [22]). An investigation of the weakest class of failure detectors to solve the non-blocking atomic commit problem when $t < n$ and there is no constraint (such as timeliness) on failure detectors can be found in [10].

Content of the paper. It has recently been shown that the failure detector class denoted $\mathcal{P}^t + \Diamond\mathcal{S}$ allows solving the consensus problem in asynchronous message-passing distributed systems prone to up to $t < n$ process crashes [8]. This means that the additional failure detection power provided by $\mathcal{P}^t$ allows suppressing the constraint $t < n/2$ (i.e.,

[1] A *timeless* failure detector does not provide information on when failures have occurred [14]. All the failure detectors considered in this paper are timeless.

the majority of correct processes assumption) to get $t < n$ (i.e., no constraint on t). $\mathcal{P}^t$ includes the failure detectors that eventually suspect the crashed processes, but at any time suspects at most $(n - t - 1)$ alive processes.

To show it, Delporte-Gallet, Fauconnier and Guerraoui have rst shown in [8] that $\mathcal{P}^t$ allows solving the *atomic register* problem. This problem consists of implementing an atomic shared variable in a message-passing system prone to process crashes. While an atomic shared variable is implementable without a failure detector when $t < n/2$ (i.e., when a majority of processes never crash) [2], it is impossible to realize it without additional assumptions when $n/2 \leq t < n$. Furthermore, it has been shown that $\diamond\mathcal{S}$ is the weakest class of failure detectors to solve consensus in asynchronous shared memory systems, and this lower bound holds for any value of $t < n$ [19]. So, combining a $\diamond\mathcal{S}$-based shared memory consensus protocol with a $\mathcal{P}^t$-based protocol building shared registers on top of a message-passing system, provides a $\mathcal{P}^t + \diamond\mathcal{S}$-based protocol solving consensus in an asynchronous message-passing system prone to up to t process crashes ($t < n$). This is the construction described in [8].

We have recently designed a $\mathcal{P}^t + \diamond\mathcal{S}$-based consensus protocol that provides a "direct" solution to the problem in the sense that it does not stack protocol layers [12]. The protocol is similar to the consensus protocol of [20], which assumes a majority of correct processes, except that in [12] we use three communication phases in each round instead of only two as in [20]. In fact, when considering these two protocols, the main difference, and in particular the added communication step in the latter one, is due to the way the *intersection property*, required to ensure agreement, is obtained. Speci cally , in [20], due to the majority of correct processes assumption, it is possible to rely on majority quorums, and obtain the intersection by a simple exchange in which each process waits to hear from a majority of other processes. On the other hand, such an exchange is not suf cient to obtain intersection with $\mathcal{P}^t$ when $n/2 \leq t < n$, and therefore an extra communication step is needed.

This paper focuses on how to obtain an intersection property with $\mathcal{P}^t$ when $n/2 \leq t < n$. It is made of two parts. The rst part generalizes the result of [12] by identifying a general two-step $\mathcal{P}^t$-based communication pattern and showing that this pattern is necessary and suf cient in order to obtain an intersection property. This pattern is demonstrated by providing a $\mathcal{P}^t + \Omega$-based consensus protocol for asynchronous message-passing systems where $t < n$. Ω is the class of *leader* oracles. It has been shown that Ω and $\diamond\mathcal{S}$ have the same computational power as far as process crash detection is concerned [4, 6]. This round-based consensus protocol is designed in a methodological manner, namely, the intersection property is used twice in each round of the protocol. It is rst used for the processes to have a weak agreement on a leader value, and then used a second time for the processes to ensure that the consensus agreement property cannot be violated. In that sense, this protocol is exemplary in the way a combination of $\mathcal{P}^t$ with a two-step communication pattern provides a noteworthy intersection property.

The second part of the paper focuses on the class of *quorum* failure detectors (denoted Σ) that has been introduced in [9]. These failure detectors provide each process with a set of trusted processes such that every two sets intersect (whatever the time at which each of them is obtained), and eventually these sets contain only correct processes. It is shown in [9], that Σ is the weakest class of failure detectors (be them realistic or not[2]) that allow solving the atomic register problem in asynchronous message passing systems where up to $t < n$ processes can crash. Σ can easily be built in systems with a majority of correct processes, i.e., when $t < n/2$. Building Σ when $n/2 \leq t < n$ requires additional power. We present in this paper a simple protocol based on $\mathcal{P}^t$ that builds such quorums while sending only constant size messages and assuming only the eventual delivery of messages sent by correct processes (yet, it does not require FIFO delivery). As $\mathcal{P}^t$ is a class of realistic failure detectors, this protocol provides a way to build realistic failure detectors of the class Σ. The protocol is designed in a methodological way from the previously identi ed two-step communication pattern. Interestingly, by showing that quorum failure detectors can be built from $\mathcal{P}^t$, this protocol participates in the proof described in [9] showing that $\mathcal{P}^t$ is the weakest class among the realistic failure detectors needed to implement an atomic register in the asynchronous distributed systems where $n/2 \leq t < n$.

Roadmap. The paper is made up of six sections. Section 2 de nes the computational model. Section 3 shows that a $\mathcal{P}^t$-based two-step communication pattern is necessary and suf cient to get the required intersection property. Section 4 then illustrates the use of the previous intersection property by presenting a $\mathcal{P}^t + \Omega$-based consensus protocol for asynchronous message-passing systems where $t < n$. Section 5 presents a $\mathcal{P}^t$-based protocol providing general quorums. Section 6 provides concluding remarks.

2 Computation Model

2.1 Asynchronous Distributed System with Process Crashes

The computation model follows the one described in [5, 11]. The system consists of a nite set Π of $n > 1$ processes, namely, $\Pi = \{p_1, \ldots, p_n\}$. A process can fail by *crashing*, *i.e.*, by prematurely halting. At most t processes can fail by crashing. A process behaves correctly (*i.e.*, according to its speci cation) until it (possibly) crashes. By de nition, a *correct* process is a process that does not crash. A *faulty* process is one that is not correct. Until it (possibly) crashes, a process is *alive*.

Processes communicate and synchronize by sending and receiving messages through channels. Every pair of processes is connected by a channel. Channels are assumed to be reliable. We say that a process "broadcasts a message" as a shortcut saying that it sends a message to each other process. This means that, while the sending of a message is atomic (a message is either sent or not sent with respect to a destination process), a broadcast is not atomic; if a process crashes while it is broadcasting a message, it is possible that some destination processes receive the message, while other processes never receive it. There is no assumption about the relative speed of processes nor on message transfer delays, i.e., the system is *asynchronous*. Let $A_{n,t}$ denote such an asynchronous message-passing system.

[2] The notion of *realistic* failure detector has been introduced in [7]. Informally, a failure detector is realistic if it can be implemented in a synchronous system. Among other features, such a failure detector cannot guess the future.

```
(1) broadcast PHASE1 $(d_i, p_i)$;
(2) wait until (PHASE1 $(-)$ messages have been received from at least $(n-t)$
                    processes and all non suspected processes);
(3) let $rec1_i$ be the set of pairs received by $p_i$ at line 2
```

Fig. 1. A $\mathcal{P}^t$-Based One-Step Communication Pattern

2.2 The Failure Detector Class $\mathcal{P}^t$

As already mentioned in the Introduction, the failure detector class $\mathcal{P}^t$ has been introduced in [8] to solve the *atomic register* problem in $A_{n,t}$ when $n/2 \leq t < n$. The class $\mathcal{P}^x$ includes all the failure detectors that satisfy the following properties:

– Strong Completeness: Eventually, every process that crashes is permanently suspected by every correct process.
– x-Accuracy: At any time, no more than $n - x - 1$ alive processes are suspected.

The reader can check that $\mathcal{P}^{n-1}$ is the class of *perfect* failure detectors (as de ned in [5]), i.e., the class of failure detectors that make no mistakes. Let us notice that $\mathcal{P}^t$ can be trivially implemented in $A_{n,t}$ when $t < n/2$. In that sense, it is only interesting to explore the properties offered by $\mathcal{P}^t$ when $n/2 \leq t < n$. Consequently, in the following, we mainly focus on asynchronous message-passing systems $A_{n,t}$ where $n/2 \leq t < n$ and equipped with a failure detector of the class $\mathcal{P}^t$.

Note that the x-Accuracy property de nes a perpetual property, while the Strong Completeness property only de nes an eventual property. This suggests that in order to implement a failure detector from the class $\mathcal{P}^t$, some form of synchrony must continuously hold in the system. For example, consider a standard heartbeat protocol for implementing $\mathcal{P}^t$. Such a protocol would work correctly if during every time interval that is equivalent to the failure suspicion timeouts, the communication links connecting any process with at least x other processes must be timely (synchronous). Clearly, this ensures that at any given moment, no process suspects more than $n - x - 1$ alive processes.

3 Implementing an Intersection Property with $\mathcal{P}^t$

3.1 $\mathcal{P}^t$ Alone Cannot Provide Set Intersection

Let us consider the one communication step protocol described in Figure 1. Each process p_i rst sends a pair (d_i, p_i) where d_i is the value it wants to disseminate, and then waits until it has received a message from all the processes it does not currently suspect and from at least $(n - t)$ processes. The set $rec1_i$ is made up of the pairs that p_i has received at the end of this communication step (i.e., when it advances to line 3).

Theorem 1. *The $\mathcal{P}^t$-based one-step communication pattern described in Figure 1 does not ensure $rec1_i \cap rec1_j \neq \emptyset$, where p_i and p_j are any pair of processes not crashed at the end of the communication phase.*

```
(1) broadcast PHASE1 $(d_i, p_i)$;
(2) wait until (PHASE1 $(-)$ messages have been received from at least $(n-t)$
                                processes and all non suspected processes);
(3) let $rec1_i$ be the set of pairs received by $p_i$ at line 2;
    ------------------------------------------------------------------------
(4) broadcast PHASE2 $(rec1_i)$;
(5) wait until (PHASE2 $(-)$ messages received from at least $(n-t)$ processes);
(6) let $rec2_i$ be the set of $(d_x, p_x)$ pairs union of the sets received at line 5;
            % We have $\forall i, j:\ rec2_i \cap rec2_j \neq \emptyset$ %
```

Fig. 2. A $\mathcal{P}^t$-Based Two-Step Communication Pattern

Proof. Let us consider a system where $n = 2t$, and partition it into two subsets P and Q, each made up of t processes (i.e., $P \cap Q = \emptyset$ and $P \cup Q = \Pi$). Let us consider the following run where $p_i \in P$ and $p_j \in Q$ are two correct processes.

- Let p_x be a process of P that crashes after it sent (d_x, p_x). Moreover, p_x is the only process of P that crashes. Similarly, let p_y be a process of Q that crashes after it send (d_y, p_y), and p_y is the only process of Q that crashes.
- While the messages within each set of processes (P and Q) arrive "quickly", the messages from each set to the other set take an arbitrarily long time to arrive (this is possible due to asynchrony).

Due to asynchrony and the t-accuracy property of $\mathcal{P}^t$, any $p_i \in P$ receives messages from all the $(n-t)$ processes in P, and suspects all the processes in Q. This is possible because $\mathcal{P}^t$ allows p_i to suspect the $(n-t-1)$ alive processes of Q. We have the same for any $p_j \in Q$. It follows that $\forall\ p_i \in P$ and $\forall\ p_j \in Q$ we have $rec1_i \cap rec1_j = \emptyset$. $\Box_{Theorem\ 1}$

This theorem shows that, in asynchronous message-passing systems $A_{n,t}$ (where $n/2 \leq t < n$) equipped with $\mathcal{P}^t$, a one-step message exchange pattern is insuf cient to provide the intersection property on the sets of values received by the processes.

3.2 A $\mathcal{P}^t$-Based Two-Step Communication Pattern

This section shows that combining $\mathcal{P}^t$ with two consecutive full information exchange phases provides the desired set intersection property. This combination is described in the protocol depicted in Figure 2 that simply adds a full information exchange phase to the one-step protocol described in Figure 1.

Theorem 2. *The $\mathcal{P}^t$-based two-step communication pattern described in Figure 2 ensures $\forall\ p_i, p_j$: $rec2_i \cap rec2_j \neq \emptyset$, where p_i and p_j are any pair of processes not crashed at the end of the second communication phase.*

Proof. Let Q_i (resp., Q_j) be the set of processes from which p_i (resp., p_j) has received a PHASE2() message (line 5). Let p_x be any process of Q_i (resp., p_y any process of

Q_j). As p_x (resp., p_y) has sent a PHASE2() message to p_i (resp., p_j), it has executed the rst phase (lines 1-3). Moreover, as a process waits at line 5 for messages from at least $(n-t)$ processes, we have $|Q_i| \geq n-t$ and $|Q_j| \geq n-t$. Let $crash_k$ denote the event "crash of p_k".

Assume (by way of contradiction) that $rec2_i \cap rec2_j = \emptyset$. We show that this cannot occur. Let e_x be the event "p_x terminates the rst phase" (i.e., the event "p_x stops waiting at line 2"). Let us observe that such an event does exist, since (by de nition) each process $p_x \in Q_i$ starts the second phase. Similarly, let p_y be any process of Q_j and e_y the event "p_y terminates the rst phase".

Due to the t-accuracy property of $\mathcal{P}^t$, when e_x occurs, p_x suspects at most $(n-t-1)$ processes that are currently alive. If p_x has received a PHASE1() message from a process $p_y \in Q_j$, we have $rec2_i \cap rec2_j \neq \emptyset$ and the theorem follows. So, let us assume that p_x has not received messages from any process $p_y \in Q_j$ when e_x occurs. As (1) p_x falsely suspects at most $(n-t-1)$ processes when e_x occurs, and (2) Q_j contains at least $(n-t)$ processes, we conclude that at least one process $p_{y'} \in Q_j$ has crashed when e_x occurs. So, there is $p_{y'} \in Q_j$ such that $crash_{y'}$ happened before e_x.

Similarly, if, for any process $p_y \in Q_j$, p_y has not received messages from any process in Q_i, we can conclude that there is at least one process $p_{x'} \in Q_i$ that has crashed before the event e_y, and $crash_{x'}$ happened before e_y.

As (1) $p_{x'} \in Q_i$, (2) $p_{y'} \in Q_j$, and (3) all the processes in Q_i and Q_j start the second communication phase, we have (from the previous observations): $crash_{x'}$ happens before $e_{y'}$ and $crash_{y'}$ happens before $e_{x'}$. Since $e_{x'}$ happens before $crash_{x'}$, and $e_{y'}$ happens before $crash_{y'}$, we get the following cycle: $e_{y'}$ happens before $crash_{y'}$ that happens before $e_{x'}$, that happens before $crash_{x'}$, that happens before $e_{y'}$. A contradiction. $\Box_{Theorem\ 2}$

Theorem 3. *In a system $A_{n,t}$ ($n/2 \leq t < n$) equipped with $\mathcal{P}^t$, two consecutive communication steps are necessary and sufficient to have $rec2_i \cap rec2_j \neq \emptyset$ where p_i and p_j are any pair of processes not crashed at the end of the second communication phase.*

Proof. Immediate consequence of Theorem 1 and Theorem 2. $\Box_{Theorem\ 3}$

4 Consensus in Asynchronous Systems Where $t < n$ with $\mathcal{P}^t + \Omega$

4.1 The Consensus Problem

As already indicated in the Introduction, in the Consensus problem, every correct process p_i *proposes* a value v_i and all correct processes have to *decide* on the same value v, that has to be one of the proposed values. More precisely, the *Consensus problem* is de ned by two safety properties (Validity and Agreement) and a Termination Property [5, 11]:

- Validity: If a process decides v, then v was proposed by some process.
- Agreement: No two processes decide differently[3].
- Termination: Every correct process eventually decides on some value.

[3] This property is sometimes called *Uniform Agreement* as it requires that a faulty process that decides, decides as a correct process.

4.2 A Leader Oracle

The failure detector class Ω is the class of *eventual leader* oracles. This class has been introduced in [4] where it is shown that it is the weakest class of failure detectors that allow solving the consensus problem in asynchronous distributed with a majority of correct processes, i.e., in $A_{n,t}$ where $t < n/2$. Ω-based consensus protocols are described in [15, 18, 21]. (Let us notice that the leader-based consensus protocol described in [18] is not explicitly expressed in terms of an underlying oracle.)

An oracle of the class Ω is a distributed device providing the processes with a function leader() whose invocations satisfy the following properties:

– Validity: Each time leader() is invoked, it returns a process name.
– Eventual Leadership: There is a time τ and a correct process p such that, after τ, every invocation of leader() by a correct process returns p.

It is important to notice that, while a leader oracle ensures that a correct leader is eventually elected, there is no knowledge of when the leader is elected. Several leaders can coexist during an arbitrary long period of time, and there is no way for the processes to learn when this "anarchy" period is over.

4.3 A $(\mathcal{P}^t + \Omega)$- Based Consensus Protocol

This section presents a $\mathcal{P}^t \times \Omega$-based protocol (Figure 3) that solves the consensus problem in asynchronous distributed systems without constraint on t (i.e., where $t < n$). A process p_i starts a consensus execution by invoking Consensus(v_i) where v_i is the value it proposes. This function is made up of two tasks, $T1$ (the main task) and $T2$. The statement *return*(v) terminates the consensus execution (as far as p_i is concerned) and returns the decided value v to p_i.

The processes proceed by consecutive asynchronous rounds. As a process that decides stops participating in the sequence of rounds and processes do not necessarily terminate in the same round, it is possible that processes proceeding to round $r + 1$ wait forever for messages from processes that decided during r. The aim of the task $T2$ is to prevent such a deadlock possibility by directing a process that decides to reliably disseminate the decided value.

As noticed in the Introduction, this protocol is designed in a methodological manner, namely, the intersection property is systematically used in each round of the protocol. It is rst used for the processes to have a weak agreement on a leader value, and then used a second time for the processes to ensure that the consensus agreement property cannot be violated. In that sense, this protocol is exemplary in the way a $\mathcal{P}^t$-based two-step communication pattern can bene t to protocol designers.

Each process p_i manages two local variables whose scope is the whole execution, namely, r_i (current round number) and est_i (current estimate of the decision value). A process p_i manages also ve local variables whose scope is the current round, namely, aux_i, and the sets of value pairs $rec1\_1_i$, $recc1\_2_i$, $rec2\_1_i$ and $rec2\_2_i$. $\perp$ denotes a

```
Function Consensus($v_i$)

Task $T1$:
(1)  $r_i \leftarrow 0$; $est_i \leftarrow v_i$;
(2)  while $true$ do
(3)      $r_i \leftarrow r_i + 1; ld_i \leftarrow$ leader(); let $d_i = (est_i, ld_i)$;

         ——————————— Phase 1 of round $r$ ———————————————-
(4)      broadcast PHASE1_1$(r_i, (d_i, p_i))$;
(5)      wait until (PHASE1_1$(r_i, (-,-))$ messages have been received from at least $n-t$
                                    processes and from all non suspected processes);
(6)      wait until (PHASE1_1$(r_i, (-,-))$ received from $p_{ld_i} \vee ld_i \neq$ leader());
(7)      let $rec1\_1_i$ be the set of $(d, p)$ pairs previously received;
(8)      broadcast PHASE1_2$(r_i, (rec1\_1_i, p_i))$;
(9)      wait until (PHASE1_2$(r_i, (-,-))$ received from at least $n-t$ processes);
(10)     let $rec1\_2_i$ be the set union of the sets of pairs previously received;
              %   Theorem 2 $\Rightarrow \forall i, j :\ rec1\_2_i \cap rec1\_2_j \neq \emptyset$   %
(11)     if $\big(\ \exists \ell : \big(\forall (d,p) \in rec1\_2_i :\ d = (-,\ell)\big) \ \wedge\ \big(((v,-), p_\ell) \in rec1\_2_i\big)\ \big)$
(12)                                           then $aux_i \leftarrow v$ else $aux_i \leftarrow \perp$ endif;
         % We have $\forall i, j :\ \big(\ (aux_i \neq \perp) \wedge (aux_j \neq \perp)\ \big)\ \Rightarrow\ (aux_i = aux_j = v)$ %
         ——————————— Phase 2 of round $r$ ———————————————-
(13)     broadcast PHASE2_1$(r_i, (aux_i, p_i))$;
(14)     wait until (PHASE2_1$(r_i, (-,-))$ messages have been received from at least $n-t$
                                    processes and from all non suspected processes);
(15)     let $rec2\_1_i$ be the set of $(aux, p)$ pairs previously received by $p_i$;
(16)     broadcast PHASE2_2$(r_i, rec2\_1_i)$;
(17)     wait until (PHASE2_2$(r_i, -)$ messages received from at least $n-t$ processes);
(18)     let $rec2\_2_i$ be the set union of the sets previously received by $p_i$;
              %   Theorem 2 $\Rightarrow \forall i, j :\ rec2\_2_i \cap rec2\_2_j \neq \emptyset$   %
(19)     let $REC_i$ be the set such that $w \in REC_i \Leftrightarrow (\exists (w,-) \in rec2\_2_i)$;
         % We have $REC_i = \{v\}$, or $REC_i = \{v, \perp\}$, or $REC_i = \{\perp\}$ %
         % Moreover, $REC_i = \{v\}$ and $REC_j = \{\perp\}$ are mutually exclusive %
(20)     case $REC_i = \{v\}$      then $est_i \leftarrow v$; broadcast DECISION($est_i$); return ($est_i$)
(21)          $REC_i = \{v, \perp\}$ then $est_i \leftarrow v$
(22)          $REC_i = \{\perp\}$     then skip
(23)     endcase
(24) endwhile

Task $T2$: when DECISION$(est)$ is received:
              do broadcast DECISION($est_i$); return ($est_i$) enddo
```

Fig. 3. A $\mathcal{P}^t + \Omega$-Based Consensus $(t < n)$

default value which cannot be proposed by processes. Moreover, each message carries the round number during which it is sent.

A round is made up of two phases. The protocol is methodologically designed: to get the intersection properties that eventually allow converging and prevent agreement violation, each phase is made up of two communication steps. More precisely, we have:

- The rst phase is a *prepare* phase. Its aim is to allow the processes to have a "quasi-agreement" on the value v of the estimate of the "current" leader. Quasi-agreement means that if two processes disagree, then at least one of them considers $\perp$ as the value of the estimate of the "current" leader. ($\perp$ can be interpreted as "I don't know".) So, the issue is for the processes to rst agree on a leader, and then adopt its value in their aux_i variables. Quasi-agreement is then

  $$\forall i,j:\ \big((aux_i \neq \perp) \wedge (aux_j \neq \perp)\big) \ \Rightarrow\ (aux_i = aux_j = v).$$

  As several leaders can coexist, the protocol has to prevent processes to consider different leaders. This is obtained with the $\mathcal{P}^t$-based two-step communication pattern described in Figure 1. During the rst step (lines 4-5) each process p_i sends a data $d_i = (est_i, ld_i)$ where ld_i is the process it currently considers as the leader.
  At the end of the second communication step (line 10), we have (Theorem 2) $\forall i,j\ :\ rec1\_2_i \cap rec1\_2_j \neq \emptyset$. This allows p_i to look if there is (possibly) a common leader, i.e., a process p_ℓ such that $\forall (d,p) \in rec1\_2_i\ :\ d = (-,\ell)$. If this predicate is true, due to the intersection property, no other process can be considered as a common leader by another process p_j. If additionally, p_i knows the value v of the current estimate of p_ℓ (predicate $((v,-),p_\ell) \in rec1\_2_i$), it adopts and keeps it in aux_i (lines 11-12).
  Hence, at the end of the rst phase, there are at most two values (namely, the current estimate v of some process and $\perp$) in the aux_i variables of the processes.
- The second phase is a *try-to-decide* phase. Its aim is to direct the processes to decide the estimate v of the common leader (if there is one), while ensuring that the consensus uniform agreement property cannot be violated when processes decide during different rounds.
  Observe that, due to the quasi-agreement provided by the rst phase, the non-$\perp$ aux_i variables are equal to the same value v. So, if two processes decide it, they trivially decide the same value. So, the main point of this phase is to prevent consensus agreement violation. This is done by ensuring that, if processes decide v during the current round, then all the processes that start the next round, do it with their estimates equal to v (hence, no value different from v can be decided in the future). This is obtained by using the intersection property once again.
  As the initial data of each process p_i is the value of aux_i, the two-step communication pattern (lines 13-19) ensures that (1) a set REC_i can be equal only to $\{v\}$, $\{v,\perp\}$ or $\{\perp\}$, and (2) (intersection property) $REC_i \cap REC_j \neq \emptyset$, from which we can conclude that, during a round, $REC_i = \{v\}$ and $REC_j = \{\perp\}$ are mutually exclusive.
  It is then easy to see that if, during a round, a process p_i decides v (line 20), then any process p_j (that neither crashes nor decides) executes line 21 and starts the next round with $est_j = v$.

4.4 Proof of the Protocol

The proof of the validity property is left to the reader.

Lemma 1. [No deadlock] *If no process decides during a round $r' \leq r$, then all correct processes start $r+1$.*

Proof. Let us assume, by way of contradiction, that no process decided during a round $r' < r$ where r is the smallest round number during which a correct process p_i blocks forever. So, p_i blocks at line 5, 6, 9, 14 or 17.

Due to the assumption on the upper bound (t) on the number of crashes, and the strong completeness property of the failure detector, it follows that no process can block forever at line 5, 9, 14 or 17.

Let us now consider line 6, and assume that p_i does not receive a PHASE1_1$(r,(-))$ message from p_{ld_i} (the process it considers as the leader). In that case, p_{ld_i} has necessarily crashed before or while it was executing line 4 (otherwise, p_i would eventually receive its PHASE1_1$(r,(-,-))$ message). Due, to eventual leadership property of the Ω oracle, there is a time after which the predicate $ld_i \neq$ leader() will become true. It follows that p_i cannot block forever at line 6. $\Box_{Lemma\ 1}$

Theorem 4. [Termination] *If a process p_i is correct, then it decides.*

Proof. Let us rst observe that if a (correct or not) process decides, then due to the fact that it broadcasts a DECISION() message before deciding (line 20 and Task $T2$) and since channels are reliable, it follows that all correct processes receive this message and decide.

So, let us assume that no process decides. The proof is by contradiction. Due to the eventual leadership property of Ω, it follows that there is a time τ after which (1) there is a correct process (say p_ℓ) such that its name ℓ is the output of all the invocations of leader(), and (2) no process crashes (i.e., all processes that execute after τ are correct). Let r be the rst round that starts after τ. As by assumption no process decides, due to Lemma 1, such a round does exist and each correct process executes it.

Let p_i be any correct process. As we are after τ, it sets $d_i = (-,\ell)$ when it starts r. Moreover, after line 6, it has received the current estimate v of p_ℓ. It follows that the predicate at line 11 is satis ed for p_i, and consequently p_i adopts v in aux_i. As during the second phase, the only value v is exchanged, it follows that each correct p_i has $REC_i = \{v\}$, and consequently decides. $\Box_{Theorem\ 4}$

Lemma 2. [Quasi-agreement] *Let p_i and p_j be any pair of processes not crashed at the end of the fir st phase of round r. We have $\big((aux_i \neq \bot) \wedge (aux_j \neq \bot) \big) \Rightarrow (aux_i = aux_j = v)$. Moreover, if $aux_i = v \neq \bot$, v is the value of the current estimate of a process that started round r.*

Proof. Let us rst observe that, as an immediate consequence of Theorem 2, we have $rec1\_2_i \cap rec1\_2_j \neq \emptyset$ after line 10 of round r. If both processes set aux_i and aux_j to $\bot$, the lemma trivially follows. So, let us consider that p_i sets aux_i to $v \neq \bot$. We show that p_j cannot set aux_j to a value different from v or $\bot$.

As p_i sets aux_i to v, the predicate it has evaluated at line 11 is true, which means that there is a process p_ℓ such that $\forall(d,p) \in rec1\_2_i : d = (-,\ell)$. As $rec1\_2_i \cap rec1\_2_j \neq \emptyset$, we conclude that $\exists(d,p) \in rec1\_2_j : d = (-,\ell)$. It follows that it is not possible for p_j to have $\forall(d,p) \in rec1\_2_i : d = (-,\ell')$ with $\ell' \neq \ell$. Consequently, due to the predicate of line 11, p_j cannot set aux_j to a value different from v or $\perp$.

The fact, if $aux_i = v \neq \perp$, v is the value of the current estimate of a process that started round r follows from the predicate evaluated at line 11. If aux_i is set to v, a pair $((v,-),p_\ell)$ has necessarily been received during that round, and consequently has been sent at the beginning of the round. $\Box_{Lemma\ 2}$

Lemma 3. [Mutual exclusion] *Let p_i and p_j be any pair of processes not crashed at line 19 of the second phase of round r. We have $REC_i = \{v\} \Leftrightarrow REC_j \neq \{\perp\}$.*

Proof. As in the previous lemma, let us rst observe that, as an immediate consequence of Theorem 2, we have $rec2\_2_i \cap rec2\_2_j \neq \emptyset$ after line 18 of round r. Moreover, due to Lemma 2, the values sent by the processes during the rst communication step of this phase are v, or v and $\perp$, or $\perp$. It follows that the sets $rec2\_2_i$ and $rec2\_2_j$ can only contain pairs such as $(v,-)$ and $(\perp,-)$. Finally, due to their de nition, the sets REC_i and REC_j can only contain v, or v and $\perp$, or $\perp$.

Let us assume that $REC_i = \{v\}$. Due to the de nition of REC_i, this means that $\forall(w,p) \in rec2\_2_i : w = v$. As $rec2\_2_i \cap rec2\_2_j \neq \emptyset$, we conclude that $\exists(w,p) \in rec2\_2_j : w = v$, which implies that $v \in REC_j$. $\Box_{Lemma\ 3}$

Theorem 5. [Agreement] *No two processes decide differently.*

Proof. Let us rst observe that a value decided by a process when executing task $T2$ comes from a process that sent it at line 20. So, we only consider values decided at line 20. Moreover, a value decided at line 20 during a round r is the value of the current estimate est_ℓ of a process that executed line 4 of r (this follows from Lemmas 2 and 3).

Let r be the smallest round during which a process p_i decides, and let v be the value it decides. This means that $REC_i = \{v\}$. Due to Lemma 3, we have, for any process p_j that executes line 19 of r, $REC_i \cap REC_j \neq \emptyset$, which means that $v \in REC_j$. It follows that if p_j decides during r it decides v. If p_j does not decide, it sets its estimate est_j to v. It follows that the estimates of all the processes that proceeds to $r+1$ are equal to v. As a value decided during a round $r' > r$ is the value of the current estimate of a process that executed line 4 of r', and as these processes have v as estimate value since the end of r, they can decide only v, and uniform agreement follows. $\Box_{Theorem\ 5}$

5 Implementing Quorum Failure Detectors

5.1 The Class of Quorum Failure Detectors

The class of *quorum* failure detectors (denoted Σ) has been introduced in [9]. At any time τ, such a failure detector provides each process p_i with a subset of processes ($trusted_i$) such that the following properties are satis ed. Let $trusted_i^\tau$ denote the value of $trusted_i$ when read by p_i at τ.

```
(1) $trusted_i \leftarrow \{p_1, \ldots, p_n\}$; $rec_i \leftarrow \{p_1, \ldots, p_n\}$; $r_i \leftarrow 0$;
(2) repeat forever
(3)     $r_i \leftarrow r_i + 1$;
(4)     broadcast ALIVE $(r_i, rec_i)$;
(5)     wait until (ALIVE $(r_i, rec)$ messages have been received from at least $(n - t)$
                                  processes and all non suspected processes);
(6)     $trusted_i \leftarrow$ union of the $rec$ sets received by $p_i$ at line 5;
(7)     $rec_i \leftarrow$ the set of proceeses from which $p_i$ received messages at line 5
(8) end repeat
```

Fig. 4. From $\mathcal{P}^t$ to Σ: A Sequence Number-Based Protocol

- Intersection: $\forall p_i, p_j : \forall \tau, \tau' : trusted_i^{\tau} \cap trusted_j^{\tau'} \neq \emptyset$.
- Completeness: Eventually, every set $trusted_i$ contains only correct processes.

Among other results, [9] presents the following theorem on the minimal failure detection requirement to implement an atomic register (this theorem considers all possible failure detectors, not only the ones that are realistic).

Theorem 6. [9] $\forall t < n$, *Σ is the weakest class of failure detectors $AS_{n,t}$ has to be equipped with in order to implement an atomic register.*

5.2 From $\mathcal{P}^t$ to Σ

This section gives a protocol that builds a failure detector of the class Σ in asynchronous distributed systems $AS_{n,t}$ with $n/2 \leq t < n$ and equipped with $\mathcal{P}^t$. (Σ can easily be built without any additional assumptions when $t < n/2$.) This protocol is constructed in an incremental way from the two step communication pattern described in Figure 2.

An incremental design. The processes need to permanently exchange messages to obtain the required intersection property. Such a permanent exchange can be realized by a repeated use of the two step communication pattern de ned in Figure 2.

Let us observe that the data part d_i is irrelevant for building quorums made up of process names. So it can be suppressed from messages. Let us now consider consecutive instances (r, $r{+}1$ and $r{+}2$) of the two step pattern. They involve the following messages (carrying their sequence number):

during	r	PHASE1 (r, p_i)	and	PHASE2 (r, rec_i)
during	$r+1$	PHASE1 $(r+1, p_i)$	and	PHASE2 $(r+1, rec_i)$
during	$r+2$	PHASE1 $(r+2, p_i)$	and	PHASE2 $(r+2, rec_i)$

We can additionally observe that a process always knows the identi er of the sender of a message it receives and that the only information carried in PHASE1(r, p_i) messages is the sender's id. Thus, for any r, each pair of messages PHASE2(r, rec_i) and PHASE1$(r+1, p_i)$ can be merged into a single message ALIVE(r, rec_i) carrying implicitly a double "PHASE1/PHASE2" semantics. When such a message is received, it is

```
(1) $trusted_i \leftarrow \{p_1,\dots,p_n\}$; $rec_i \leftarrow \{p_1,\dots,p_n\}$;
(2) repeat forever
(3)     broadcast ALIVE $(rec_i)$;
(4)     wait until (new ALIVE $(rec)$ messages have been received from at least $(n-t)$
                     processes and all non suspected processes);
(5)     $trusted_i \leftarrow$ union of the $rec$ sets received by $p_i$ at line 4;
(6)     $rec_i \leftarrow$ the set of processes from which $p_i$ received messages at line 4
(7) end repeat
```

Fig. 5. From $\mathcal{P}^t$ to Σ: An Ef cient Protocol

processed according to its double semantics. In that way, we get the protocol described in Figure 4 that implements Σ from $\mathcal{P}^t$.

Note that each iteration in the protocol of Figure 4 consists of only one message exchange, which at rst glance appears to contradict the necessity of having two communication steps. A closer look, however, reveals that there in no contradiction. Essentially, the value of *trusted*$_i$ is only valid after the second iteration of the protocol. Thus, consider any implementation of the application interface of $\mathcal{P}^t$, which returns the list of trusted processes to an application or higher level protocol. Such an interface implementation must report all processes as trusted until at least the second iteration of the protocol in Figure 4 terminates. Only after the second iteration terminates it can return the variable *trusted*$_i$.

The fnal protocol. A close look at the previous protocol reveals that the sequence numbers are actually useless. Intuitively, this “comes” from the fact that the intersection property we want to obtain states that any two sets of trusted processes have to intersect whatever the time at which they have been computed (e.g., even if one has been computed at the very begining of the computation, while the other has been computed much later). The suppression of sequence numbers provides a more concise protocol in which all messages have a bounded size. The ef cient protocol resulting from this observation is described in Figure 5. Let us notice that this protocol only requires eventual delivery of messages sent by correct processes, yet it does not rely on FIFO.

5.3 Proof of the Protocol

While the protocol described in Figure 5 does not associate sequence numbers with iteration steps, for the sake of the proof, we identify each iteration executed by a process by an *iteration number*. Moreover, the proof uses the following notation. For any iteration number k and process p_i not crashed at the end of k:

- r_i^k denotes the execution of the k-th iteration of p_i,
- $trusted_i^k$ denotes the value of the set $trusted_i$ computed by p_i at the end of its k-th iteration.

Lemma 4. [No deadlock] *Every correct process executes line 4 infinitely often.*

Proof. Assume, by way of contradiction, that some correct process blocks during the execution of one of its protocol's iterations. Without loss of generality, let p_i be the process that blocks on the smallest iteration number (if there are several such processes, then p_i is the one with smallest id), and denote this iteration number by k.

By examining the code of the protocol, the only place where a process could block is the wait statement of line 4. In this statement a process waits until is received messages from all unsuspected processes and from at least $n - t$ processes. Yet, due to the completeness property of $\mathcal{P}^t$, all crashed processes are eventually detected, and thus, a process cannot wait forever due to a failed process.

Moreover, by the assumption on the minimality of k, then all correct processes eventually start their k-th iteration, and in particular, each of them sends at least k distinct ALIVE$(-)$ messages. Thus, as there are at most t failures, at least $n - f$ new messages should be received by p_i during r_i^k. Therefore, p_i cannot block at line 4, and consequently proceeds to the next iteration and starts r_i^{k+1}. A contradiction. $\Box_{Lemma\ 4}$

Lemma 5. [Completeness] *The protocol described in Figure 5 ensures Σ completeness property.*

Proof. When a process p_i crashes, it stops sending new messages. Let τ be the time after which all the messages it has sent have been received. Clearly, no new messages from p_i are received after τ. By Lemma 4, each correct process executes the protocol's iterations in nitely often. Thus, it follows from the code of Lines 5 and 6 (and the strong completeness property of $\mathcal{P}^t$) that every correct process will permanently suspect p_i some nite time after τ. $\Box_{Lemma\ 5}$

Lemma 6. [Intersection] *Let p_i (resp., p_j) be a process that executes iteration number k (resp., number l). The protocol described in Fig. 5 ensures $trusted_i^k \cap trusted_j^l \neq \emptyset$.*

Proof. Let us rst observe that we can conclude from the protocol's code that, for any p_x and any iteration ℓ, the set $trusted_x^\ell$ cannot be empty. Let us also notice that if, during its k-th iteration, p_i receives a message that its sender p_x sent during its rst iteration, it follows from the initial value of rec_x and line 5 that $trusted_i^k = \{p_1, \ldots, p_n\}$, and consequently $trusted_i^k \cap trusted_j^l \neq \emptyset$. Thus, for the rest of the proof, we only consider cases where neither p_i nor p_j receives any message that was sent by a process in its rst protocol's iteration.

Assume, by way of contradiction, that there are two processes p_i and p_j and two respective iteration numbers k and l for which $trusted_i^k \cap trusted_j^l = \emptyset$. Let Q_i^k (resp., Q_j^l) be the set of processes from which p_i (resp., p_j) has received an ALIVE$(-)$ message (line 4) during r_i^k (resp., r_j^l). Let p_x be a process of Q_i^k (resp., p_y a process of Q_j^l), and consider the iteration number k_x (resp., l_y) in which p_x (resp., p_y) sent this message. Moreover, without loss of generality, let us assume that, among the processes in Q_i^k, p_x is the rst that terminated the iteration (here $k_x - 1$) just before the one (namely, k_x) during which it sent the ALIVE$(-)$ message that p_i has received during its k-th iteration. Similarly, let us assume that, among the processes in Q_j^l, p_y is the rst that terminated the iteration (here $k_l - 1$) just before the one (namely, k_l) during which it sent the ALIVE$(-)$ message that p_j has received during its l-th iteration. Note that the values of k, l, k_x, and l_y can be completely distinct from each other, but can also be the same; the proof does not depend on this.

As p_x (resp., p_y) has sent an ALIVE$(-)$ message to p_i (resp., p_j) during $r_x^{k_x}$ (resp., $r_y^{l_y}$), it has nished executing its iteration number $k_x - 1$ (resp., number $l_y - 1$) by the time it sends this ALIVE$(-)$ message. Moreover, as a process waits at line 4 for messages from at least $(n-t)$ processes, we have $|Q_i^k| \geq n-t$ and $|Q_j^l| \geq n-t$.

Let $crash_q$ denote the event "crash of p_q" for any process p_q. Also, let e_x be the event "p_x terminates the iteration $k_x - 1$", and, similarly, e_y be the event "p_y terminates its iteration $l_y - 1$".

Due to the t-accuracy property of $\mathcal{P}^t$, when e_x occurs, p_x suspects at most $(n-t-1)$ processes that are currently alive. If p_x has received an ALIVE$(-)$ message from a process $p_y \in Q_j^l$ during $r_x^{k_x}$, then by the code we have $trusted_i^k \cap trusted_j^l \neq \emptyset$ and the theorem follows. So, let us assume that p_x has not received messages from any process $p_y \in Q_j^l$ during $r_x^{k_x-1}$, (the iteration at the end of which the event e_x occurs). As (1) p_x falsely suspects at most $(n-t-1)$ processes when e_x occurs, and (2) Q_j^l contains at least $(n-t)$ processes, it follows that at least one process $p_{y'} \in Q_j^l$ has crashed when e_x occurs. So, there is $p_{y'} \in Q_j^l$ such that $crash_{y'}$ happened before e_x.

Similarly, if, for any process $p_y \in Q_j^l$, p_y has not received messages from any process in Q_i^k during $r_y^{l_y-1}$, we can conclude that there is at least one process $p_{x'} \in Q_i^k$ that has crashed before the event e_y, or in other words, $crash_{x'}$ happened before e_y.

As $p_{x'} \in Q_i^k$ and $p_{y'} \in Q_j^l$, we have the following "happened before" ordering from the previous observations.

- $crash_{x'}$ happens before e_y that happens before $e_{y'}$ (e_y happens before $e_{y'}$ because, by assumption, p_y is the rst process of Q_j^l that nished executing its iteration preceding the one during which it sent the ALIVE() message received by p_j during r_j^l).
- $crash_{y'}$ happens before e_x that happens before $e_{x'}$ (similarly, e_x happens before $e_{x'}$ because, by assumption, p_x is the rst process of Q_i^k that nished executing its iteration preceding the one during which it sent the ALIVE() message received by p_i during r_i^k).

Since $e_{x'}$ happens before $crash_{x'}$, and $e_{y'}$ happens before $crash_{y'}$, we get the following cycle: $e_{y'}$ happens before $crash_{y'}$, which happens before $e_{x'}$, which happens before $crash_{x'}$, which happens before $e_{y'}$. A contradiction. $\Box_{Lemma\ 6}$

Theorem 7. *The protocol described in Figure 5 builds a failure detector of the class Σ in any asynchronous distributed system made up of n processes where up to t can crash, equipped with a failure detector of the class $\mathcal{P}^t$.*

Proof. The theorem follows directly from the Lemmas 5 and 6. $\Box_{Theorem\ 7}$

6 Concluding Remarks

On weakest realistic failure detectors The theorem below complements the results of [9] on the weakest class of realistic failure detectors for the atomic register problem.

Theorem 8. *$\mathcal{P}^t$ is the weakest class of realistic failure detectors that allows solving the atomic register problem in $AS_{n,t}$ where $t < n$.*

Proof. The theorem follows from the following three facts. (1) It is shown in [9] that any realistic failure detector in Σ is in $\mathcal{P}^t$. (2) The previous theorem 7 has shown that $\mathcal{P}^t$ allows the construction of realistic failure detectors of Σ. (3) It is shown in [9] that Σ is the weakest class of failure detectors $AS_{n,t}$ has to be equipped with in order to implement an atomic register (theorem 6). $\square_{Theorem\ 8}$

On failure detector reduction protocols. We have designed in [1] a protocol that builds a failure detector of the class $\diamond\mathcal{S}$ from an underlying failure detector denoted $\diamond\mathcal{S}_k$ whose accuracy property is "weaker" than the one of $\diamond\mathcal{S}$ in the following sense. Accuracy of $\diamond\mathcal{S}$ requires that there is a correct process that is eventually not suspected by the other processes. Hence, the scope of the accuracy property is n as it is a priori on all the processes. The accuracy of $\diamond\mathcal{S}_k$ requires that there is a correct process that is eventually not suspected by a set of k other processes. When $k = n$, both properties are the same.

We have shown in [1] that there is a protocol building $\diamond\mathcal{S}$ from $\diamond\mathcal{S}_k$ if an only if $k > t$. It is noteworthy to notice that although it was derived from different principles, the structure of the protocol proposed in [1] that reduces $\diamond\mathcal{S}$ to $\diamond\mathcal{S}_k$ when $k > t$ is the same as the nal structure of the protocol described in Figure 5 (but, of course, with different inputs and different outputs). This shows that some failure detector reduction protocols exhibit an interesting unity that may merit additional investigation.

Communication vs failure detection. As a nal remark, while quorums are used to convey data and communicate values among the processes, they provide no information on failures. Differently, a failure detector provides (sometimes unreliable) information on failures but does not allow the processes to communicate information. Yet, combining failure detectors with quorums allows the processes to progress and terminate in a consistent way (i.e., while ensuring uniform agreement). More precisely, in the approach followed in the consensus protocol we have presented:

– Consensus termination is ensured thanks to the $\Omega/\diamond\mathcal{S}$ part of the failure detector.
– Uniform agreement is ensured thanks to the quorum intersection property, that is realized by a $\mathcal{P}^t$-based two-step communication pattern.

The second item indicates that a failure detector such as $\mathcal{P}^t$ provides enough synchronization power to enable a communication pattern that can convey information among processes whatever the value of t. This means that some failure detectors can be used to augment the power of communication primitives.

Acknowledgments. We would like to thank Carole Delporte, Hugues Fauconnier and Rachid Guerraoui for interesting discussions on the failure detector classes $\mathcal{P}^t$ and Σ.

References

1. Anceaume E., Fernandez A., Mostéfaoui A., Neiger G. and Raynal M., Necessary and Suf - cient Condition for Transforming Limited Accuracy Failure Detectors. *Journal of Computer and System Science (JCSS)*, 68:123-133, 2004.

2. Attiya H., Bar-Noy A. and Dolev D., Sharing Memory Robustly in Message Passing Systems. *Journal of the ACM*, 42(1):121-132, 1995.
3. Bermond J.-Cl., Koenig J.-Cl. and Raynal M., Generalized and Ef cient Decentralized Consensus Protocols. *Proc. 2nd Int. Workshop on Distributed Algorithms and Graphs (WDAG'87, now DISC)*, Springer-Verlag LNCS #312, pp. 41-56, 1987.
4. Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685–722, 1996.
5. Chandra T.D. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.
6. Chu F., Reducing Ω to $\diamond\mathcal{W}$. *Information Processing Letters*, 76(6):293-298, 1998.
7. Delporte-Gallet C., Fauconnier H. and Guerraoui R., A Realistic Look at Failure Detectors. *Proc. Int. Conference on Dependable Systems and Networks (DSN'02)*, pp. 354-353, 2002.
8. Delporte-Gallet C., Fauconnier H. and Guerraoui R., Failure Detection Lower Bounds on Registers and Consensus. *Proc. 16th Int. Symposium on Distributed Computing (DISC'02)*, Springer-Verlag LNCS #2508, pp. 237-251, 2002.
9. Delporte-Gallet C., Fauconnier H. and Guerraoui R., Shared Memory vs Message Passing. *Technical Report*, IC/2003/77, EPFL, Lausanne (Switzerland), 2003.
10. Delporte-Gallet C., Fauconnier H., Guerraoui R., Hadzilacos V., Kouznetzov P. and Toueg S., The Weakest Failure Detectors to Solve Certain Fundamental Problems in Distributed Computing. *23th ACM Int. Symp. on Principles of Distributed Computing*, pp. 338-346, 2004.
11. Fischer M.J., Lynch N. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.
12. Friedman R., Mostéfaoui A. and Raynal M., A Weakest Failure Detector-Based Asynchronous Consensus Protocol for $f < n$. *Information Processing Letters*, 90(1):39-46, 2004.
13. Guerraoui R., Non-Blocking Atomic Commit in Asynchronous Distributed Systems with Failure Detectors. *Distributed Computing*, 15:17-25, 2002.
14. Guerraoui R. and Kouznetsov P., On the Weakest Failure Detector for Non-Blocking Atomic Commit. *Proc. 2nd Int. IFIP Conf. on Theoretical Computer Science*, pp. 461-473, 2002.
15. Guerraoui R. and Raynal M., The Information Structure of Indulgent Consensus. *IEEE Transactions on Computers.* 53(4), 53(4):453-466, April 2004.
16. Keidar I. and Dolev D., Increasing the Resilience of Distributed and Replicated Database Systems. *Journal of Computer and System Sciences*, 57(3):309-324, 1998.
17. Lakshman T.V. and Agrawala A.K., Ef cient Decentralized Consensus Protocols. *IEEE Transactions on Software Engineering*, SE12(5):600-607, 1986.
18. Lamport L., The Part-Time Parliament. *ACM TOCS*, 16(2):133-169, 1998.
19. Lo W.-K. and Hadzilacos V., Using Failure Detectors to Solve Consensus in Asynchronous Shared-Memory Systems. *Proc. 8th Int. Workshop on Distributed Algorithms (WDAG'94)*, Springer-Verlag LNCS #857, pp. 280-295, 1994.
20. Mostéfaoui A. and Raynal M., Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: a General Quorum-Based Approach. *Proc. 13th Int. Symposium on Distributed Computing (DISC'99)*, Springer-Verlag LNCS #1693, pp. 49-63, 1999.
21. Mostéfaoui A. and Raynal M., Leader-Based Consensus. *Parallel Processing Letters*, 11(1):95-107, 2001.
22. Skeen D., Non-Blocking Commit Protocols. *Proc. ACM SIGMOD Int. Conference on Management of Data*, ACM Press, pp. 133-142, 1981.
23. Thomas R.H., A Majority Consensus Approach to Concurrency Control for Multiple Copies Databases. *ACM Transactions on Database Systems*, 4(2):180-209, 1979.

Failure Detection with Booting in Partially Synchronous Systems[⋆]

Josef Widder[1], Gérard Le Lann[2], and Ulrich Schmid[1]

[1] Technische Universität Wien, Embedded Computing Systems Group E182/2, Treitlstraße 3, A-1040 Vienna (Austria)
{widder, s}@ecs.tuwien.ac.at

[2] INRIA Rocquencourt, Projet Novaltis, BP 105, F-78153 Le Chesnay Cedex (France)
gerard.le_lann@inria.fr

Abstract. Unreliable failure detectors are a well known means to enrich asynchronous distributed systems with time-free semantics that allow to solve consensus in the presence of crash failures. Implementing unreliable failure detectors requires a system that provides some synchrony, typically an upper bound on end-to-end message delays. Recently, we introduced an implementation of the perfect failure detector in a novel partially synchronous model, referred to as the Θ-Model, where only the ratio Θ of maximum vs. minimum end-to-end delay of messages that are simultaneously in transit must be known a priori (while the actual delays need not be known and not even be bounded). In this paper, we present an alternative failure detector algorithm, which is based on a clock synchronization algorithm for the Θ-Model. It not only surpasses our first implementation with respect to failure detection time, but also works during the system booting phase.

1 Introduction

Asynchronous distributed algorithms maximize systems coverage, i.e., the probability that a fault-tolerant distributed real-time system works as required during its lifetime. In particular, systems coverage is known to be higher with asynchronous algorithms than with synchronous algorithms, for identical performance figures (e.g., response times) [1, 2]. Due to the well-known FLP impossibility result [3], however, important generic problems like consensus cannot be solved deterministically in purely asynchronous systems if just a single process may crash. Solving such generic problems hence requires a purely asynchronous system augmented with some semantics.

Dolev, Dwork and Stockmeyer [4] investigated how much synchronism is required in order to solve consensus. They identified five synchrony parameters

[⋆] Supported by the Austrian START program Y41-MAT, the BM:vit FIT-IT project *DCBA* (proj. no. 808198), and by the FWF project *Theta* (proj. no. P17757-N04).

(processors, communication, message order, broadcast facilities and atomicity of actions), which can be varied into 32 different [illegible] models. For each of those, they investigated whether consensus is solvable. A different — more abstract — approach was taken by Chandra and Toueg [5,6], who introduced the concept of unreliable [illegible] (FDs). A local failure detector module is associated with every process, which provides the consensus algorithm with hints about processes that (seem to) have crashed. Several different classes of unreliable failure detectors sufficient for solving consensus have been identified in [5]. In this paper we focus on the perfect FD $\mathcal{P}$. Informally, this FD has the semantics that (1) all correct processes will detect all crashes and (2) no processes will be falsely suspected of having crashed.

Failure detectors are particularly attractive, since they encapsulate synchrony assumptions in a time-free manner. Consensus algorithms using FDs are hence time-free in the sense that no local clocks are needed and no real-time variables show up in the algorithm's code. Therefore such algorithms share the coverage maximization property proper to purely asynchronous algorithms. Nevertheless, the question of coverage arises also when [illegible] an FD, which of course requires the underlying system to satisfy some synchrony assumptions. In fact, existing implementations of the perfect FD $\mathcal{P}$ (see [7] for a comprehensive overview of existing work) rest upon knowing an upper bound on the end-to-end transmission delays of messages and hence require a synchronous system [8]. In [7] and [2], we introduced a novel system model, referred to as Θ-Model[1], which is essentially the purely asynchronous FLP model [3] augmented with a bound upon the ratio Θ of maximum vs. minimum end-to-end computation + transmission delay between correct processes. Since just a bound upon the ratio Θ — but not on the maximum delay itself — must be known, this type of partial synchrony is not covered in the classic literature on synchrony [4,9,8]: The existing models require both an upper bound upon (1) the relative speed Φ of any two correct processes and (2) the absolute message transmission delay Δ. In sharp contrast, the Θ-Model does not incorporate any absolute bound on delays and works even in situations where actual delays are unbounded. (This is formalized in [10].) Unlike the global stabilization time model of [9], which assumes that the system is synchronous from some unknown point in time on, it belongs to the class of models where bounds on transmission and computation delays are unknown but are always assumed to hold.

Another issue where the Θ-Model differs from the partially synchronous models of [4,9,8] is the fact that it allows message-driven algorithms only, where every computation event at a process is a direct response to some (remote) message reception. In [7] we showed that spontaneous local events — e.g. due to clocks or timers — are in fact not necessary for solving generic agreement problems. Θ-algorithms are hence time-free in that they do not rely upon local time information (no clocks, no a priori bounds on the duration of computation steps) and can only make decisions based on and triggered by received messages.

[1] Visit *http://www.ecs.tuwien.ac.at/projects/Theta/* for our papers on the Θ-Model.

In [7], we also introduced an algorithm for implementing $\mathcal{P}$ in the Θ-Model. By definition, this algorithm circumvents both the impossibility of implementing $\mathcal{P}$ in the presence of [illegible] delay bounds of [11] and the impossibility of consensus in presence of [illegible] delays of [4], by resorting to the assumption of an a priori known ratio between largest and shortest end-to-end delays.

This paper presents an alternative implementation of $\mathcal{P}$ in the Θ-Model, which surpasses the failure detector implementation [7] with respect to detection time. The new solution is based upon a clock synchronization algorithm introduced in [12, 13], which employs an extension of the non-authenticated clock synchronization algorithm by Srikanth and Toueg [14]. Among its particularly attractive features is its ability to properly handle system booting: Given that the Θ-Model allows just message-driven algorithms, Θ-algorithms must implement any required functionality without resorting to time or timers. This also constrains the solution space for the important — but often neglected — system startup problem considerably: Since our FD algorithm requires a quorum of processes in order to achieve its properties, we cannot simply assume that it works also during the booting phase, where processes get up independently of each other at unpredictable times. After all, processes that get up late typically miss at least some of the messages sent by earlier started ones.

Straightforward system startup solutions (see [12] for an overview of existing approaches) either constrain the maximum allowed duration of the booting phase via timeouts, or considerably increase the system size n for a given number of failures to be tolerated. Both deficiencies are avoided by the clock synchronization algorithm of [12], which in fact guarantees some of its properties (including precision) also during system startup. Using this algorithm, we provide an implementation of $\mathcal{P}$ for the Θ-Model which behaves like an eventually perfect failure detector $\diamond\mathcal{P}$ during system startup and becomes $\mathcal{P}$ when sufficiently many processes have completed booting.

[illegible] After an overview and a short discussion of the Θ-Model in Section 2 and Section 3, respectively, we provide some required results related to the clock synchronization algorithm of [12, 13] in Section 4. Section 5 starts with the presentation of the new FD algorithm for the simplified case where all processes boot simultaneously, which is then extended to handle realistic system startup scenarios. A short discussion of our results and some remarks on coverage issues in Section 6 complete the paper.

2 System Model

We consider an asynchronous distributed system of n processes denoted by $p, q, \ldots$, which communicate through a reliable, error-free and fully connected point-to-point network. Even under the perfect communication assumption, messages that reach a process that is not booted are lost. The communication channels do not necessarily provide FIFO delivery of messages. We assume that every (non-faulty) receiver of a message knows its sender, which is actually ensured by our point-to-point assumption. Our algorithms do not require an authentication service. (Except when our algorithms have to be implemented in systems where a point-to-point network must be simulated via a shared channel like Ethernet, since authentication is required to prevent masquerading here.)

[illegible] Among the n processes, there is a maximum of f faulty ones. When considering just the clock synchronization algorithm in Section 4, no restriction is put on the behavior of faulty processes; they may exhibit Byzantine failures. Since the existing work on muteness detector specifications [15, 16, 17, 18, 19] suggests to also consider more severe types of failures in FD-based applications, we decided to retain the Byzantine failure model for the implementation of our FD as well. More advanced hybrid versions of our algorithm, which tolerate hybrid processor and link failures, can be found in [2, 13]. Note that the perfect communications assumption could also be dropped in favor of fair lossy links by using the simulation of reliable links proposed by Basu, Charron-Bost and Toueg [20]. Since we also investigate system startup, correct processes that have not booted yet are not counted as faulty.

Despite of using the Byzantine failure model for implementing our FD, it is nevertheless true that the classic perfect failure detector specification is only meaningful for crash failures. When our FD is used in conjunction with a classic FD-based consensus algorithm [5], the f faulty processes should hence exhibit crash failures only. Actually, even a classic FD-based consensus algorithm using our FD would also tolerate early timing failures, i.e., up to f processes that inconsistently output correct messages too early.

[illegible] Let [illegible] be the abstraction/implementation level of our FD and the underlying clock synchronization algorithm. Following the [illegible] approach of [1], this level should typically be thought of as a level fairly close to the raw computing and communication facilities. In contrast, such algorithms as consensus or atomic broadcast are typically thought of as middleware-level algorithms. Combining this with appropriate scheduling algorithms, it follows that FD-level end-to-end delays are significantly smaller than those end-to-end delays proper to consensus or atomic broadcast algorithms — typically, one order of magnitude smaller (see [1]). This feature is of particular interest with the Θ-Model [7], which is the computational model employed in this paper. For simplicity, we employ the basic version of the Θ-Model [7] here: Assume that the FD-level end-to-end delay δ_{pq} between any two correct processes p and q satisfies $\tau^- \leq \delta_{pq} \leq \tau^+$, where the maximum and minimum $\tau^+ < \infty$ and $\tau^- > 0$, respectively, are not known a priori. δ_{pq} includes computation delays at sender and receiver, communication delays, and sojourn times in waiting queues. Note that $\tau^- > 0$ must also capture the case $p = q$ here; $\tau^+ < \infty$ secures that every message is eventually delivered. The timing uncertainty is determined by the [illegible] $\varepsilon = \tau^+ - \tau^-$ and the [illegible] $\Theta = \tau^+/\tau^-$. Note that neither τ^- nor τ^+ show up in our algorithm's code, but only Ξ, which is a function of an a priori given[2] upper bound $\bar{\Theta}$ upon Θ. (We discuss some consequences of the fact that just $\bar{\Theta}$ needs to be known a priori in Section 3.)

In [10], it has been shown formally that for ensuring safety and liveness properties of Θ-algorithms, τ^+ and τ^- need not even be invariant, i.e., that they may vary during operation. The only requirement is that they increase and decrease together such that Θ always holds. Note that this time-variance prohibits to compute a valid upper bound on τ^+ as the product of some measured message delay and Θ, since this bound may already be invalid when it is eventually applied. A general theorem allows algorithms to be analyzed for constant τ^+ and τ^- (like it is done in this paper), however, and the results — e.g. Theorem 5 and Theorem 7 — to be translated directly into the variable timing model of [10].

Note finally that there are more advanced versions of the Θ-Model, which allow a more accurate (i.e., less pessimistic e.g. w.r.t. detection time) modeling of the behavior of our FD algorithm in real systems. Lacking space does not allow us to elaborate on those extensions here.

[illegible] Initially, all correct processes are down, i.e., do not send or receive messages. Every message that arrives at a correct process while it is down is lost. A correct process decides independently when it wishes to participate in the system (or is just switched on). As faulty processes may be Byzantine, we can safely assume that faulty processes are always up or at least booted before the first correct one. Correct processes go through the following operation modes:

1. [illegible]: A process is down when it has not been started yet or has not completed booting.
2. [illegible]: A process is up if it has completed booting. To get a clean distinction of up and down, we assume that a process flushes the input queues of its network interface as first action after booting is completed. Hence, it receives messages only if they have arrived when it was up.

3 Discussion of the Θ-Model

In this section, we provide a short justification and discussion of the Θ-Model taken from [7]. Our arguments will show why a timing model where a bound on message delays is replaced by a bound on the ratio of largest and shortest end-to-end message delays makes sense for distributed fault-tolerant real-time systems.

In real systems, the end-to-end message delay δ_{pq} consists not only of physical data transmission and processing times. Rather, [illegible] due to the inevitable [illegible] of the concurrent execution of multiple processes and message arrival interrupts/threads on every processor must be added to the picture.

[2] Overbars are used for given bounds ($\bar{\Theta}$) on actual values (Θ). Such bounds must be derived from worst case and best case schedulability analyses.

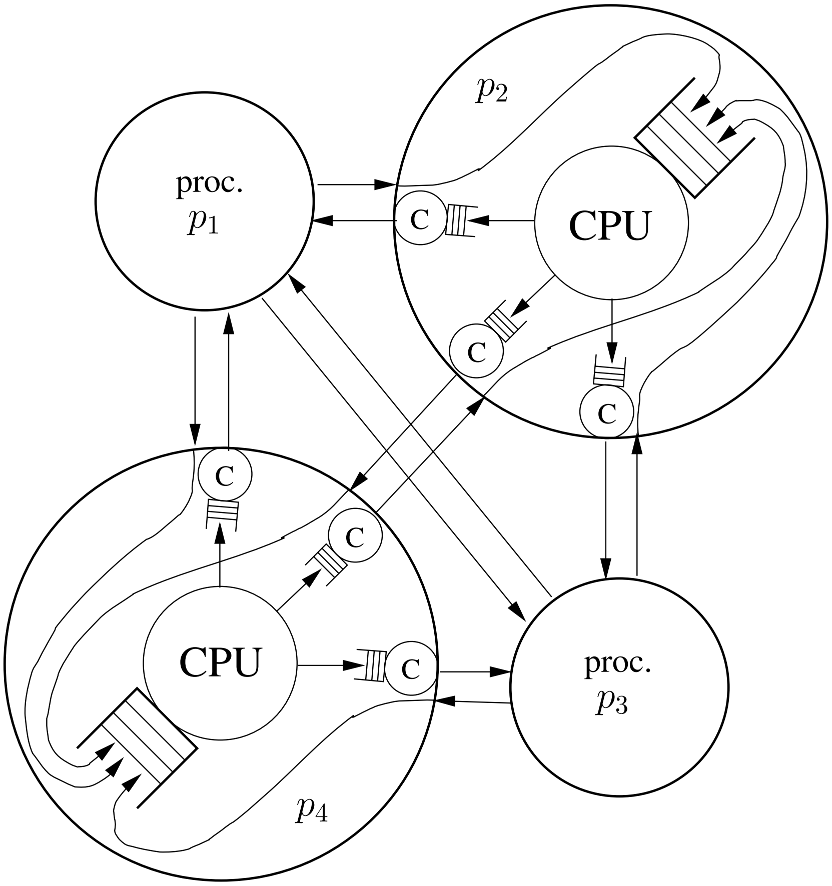


Fig. 1. A simple queuing system representation of a fully connected distributed system

Figure 1 shows a simple queuing system model of a fully connected distributed system: All messages that drop in over one of the $n-1$ incoming links of a processor must eventually be processed by the single CPU. Every message that arrives while the CPU processes former ones must hence be put into the CPU queue for later processing. In addition, all messages produced by the CPU must be scheduled for transmission over every outgoing link. Messages that find an outgoing link busy must hence be put into the send queue of the link's communication controller for later transmission.

Consequently, the end-to-end delay $\delta_{pq} = d_{pq} + \omega_{pq}$ between sender p and receiver q consists of a "fixed" part d_{pq} and a "variable" part ω_{pq}. The fixed part $d_{pq} > 0$ is solely determined by the processing speeds of p and q and the data transmission characteristics (distance, speed, etc.) of the interconnecting link. It determines the minimal conceivable δ_{pq} and is easily determined from the physical characteristics of the system. The real challenge is the variable part $\omega_{pq} \geq 0$ which captures all scheduling-related variations of the end-to-end delay:

- Precedences, resource sharing and contention, with or without resource preemption, which creates waiting queues,
- Varying (application-induced) load,
- Varying process execution times (which may depend on actual values of process variables and message contents),
- Occurrence of failures.

It is apparent that ω_{pq} and thus δ_{pq} depend critically upon (1) the scheduling strategy employed (determining which message is put at which place in a queue), and (2) the particular distributed algorithm(s) executed in the system: If the usual FIFO scheduling is replaced by head-of-the-line scheduling favoring FD-level messages and computations over all application-level ones, as done with the fast failure detectors of [1], the variability of ω_{pq} at the FD-level can be decreased by orders of magnitude, see Table 1. That the queue sizes and hence the end-to-end delays δ_{pq} increase with the number and processing requirements of the messages sent by the particular distributed algorithm that is run atop of the system is immediately evident. Interestingly, however, fast FDs diminish the effect of the latter upon FD-level end-to-end delays as well, as the highest priority processing and communication activities involved with FDs are essentially "adversary immune" (the "adversary" being all activities other than FD-level related ones, in particular, application-level ones) here. This reduces both the order of magnitude of δ_{pq} and the complexity of schedulability analyses (see below) very significantly.

The above queuing system model thus reveals that the popular synchronous and partially synchronous models rest upon a very strong assumption: That an a priori given upper bound B exists which is — as part of the model — essentially independent of the particular algorithm or service under consideration, independent of the scheduling algorithm(s) used, as well as independent of the "loads" generated by algorithms or services other than the one being considered. Since the distinction between FD-level and application level is almost never made, it is almost always the case that $B \gg \bar{\tau}^+ \geq \delta_{pq}$.

In reality, such a bound B can only be determined by a detailed [illegible][3] [21, 22]. In order to deal with all the causes of delays listed above, this schedulability analysis requires complete knowledge of the underlying system, the scheduling strategies, the failure models, the failure occurrence models and, last but not least, the particular algorithms that are to be

[3] Measurement-based approaches are a posteriori solutions. Asserting a priori knowledge of an upper bound implies predictability, which is achievable only via worst-case schedulability analysis. With measurement-based approaches, the actual bounds remain unknown (even "a posteriori"), which might suffice for non-critical systems, but is out of question with many real-time embedded systems, safety-critical systems in particular.

executed in the system. Compiling B into the latter algorithms, as required by solutions that rest upon timing assumptions, hence generates a cyclic dependency. Moreover, conducting a detailed worst-case schedulability analysis for a solution that is not "adversary immune" is notoriously difficult. Almost inevitably, it rests on simplified models of reality (environments, technology) that may not always hold. As a consequence, B and hence any non time-free solution's basic assumptions might be violated at run time in certain situations.

The actual value of $\bar{\Theta}$, which obviously depends upon many system parameters, can only be determined by a detailed schedulability analysis as well. Note, however, that $\bar{\tau}^+$ has a coverage which can only be greater than the coverage of B, since our design for FDs is "adversary immune". Note also that $\bar{\tau}^-$ which is the result of a best case analysis (just d_{pq}) can be considered to have an assumption coverage higher than $\bar{\tau}^+$.

Values for $\bar{\Theta}$ depend of the physical properties of the system considered. They may range from values close to 1 in systems based on satellite broadcast channels to higher values in systems where processing and queuing delays dominate propagation delays.

To get an idea of how $\Theta(t)$ behaves in a real system ($\Theta(t)$ is the relation of longest and shortest end-to-end delays just of the messages that are simultaneously in transit at time t), we conducted some experiments [23] on a network of Linux workstation running our FD algorithm. A custom monitoring software was used to determine bounds $\bar{\tau}^- \leq \tau^-(t)$ and $\bar{\tau}^+ \geq \tau^+(t)$ as well as $\bar{\Theta} \geq \Theta(t)$ under a variety of operating conditions. The FD algorithm was run at the application level (AL-FD), as an ordinary Linux process, and as a fast failure detector (F-FD) using high-priority threads and head-of-the-line scheduling [1]. Table 1 shows some measurement data for 5 machines (with at most $f = 1$ arbitrary faulty one) under low (5 % network load) and medium (0–60 % network load, varying in steps) application-induced load.

Table 1. Some typical experimental data from a small network of Linux workstations running our FD algorithm

FD	**Load**	$\bar{\tau}^-$ **(μs)**	$\bar{\tau}^+$ **(μs)**	$\bar{\Theta}$	$\frac{\bar{\tau}^+}{\bar{\tau}^-} : \bar{\Theta}$
AL-FD	low	55	15080	228.1	1.20
F-FD	low	54	648	9.5	1.26
F-FD	med	56	780	10.9	1.27

Our experimental data thus reveal that a considerable correlation between $\tau^+(t)$ and $\tau^-(t)$ indeed exists: The last column in Table 1 shows that $\bar{\Theta}$ is nearly 30 % smaller than $\bar{\tau}^+/\bar{\tau}^-$. Hence, when τ^+ increases, τ^- goes up to some extent as well. Note that τ^- must in fact only increase by α/Θ to compensate an increase of τ^+ by α without violating the Θ-assumption.

Informally, this correlation between τ^+ and τ^- for systems where all communication is by (real or simulated) broadcasting can be explained as follows: If some message m experiences a delay greater than τ^+, this is due to messages scheduled ahead of it in the queues along the path from sender p to q. Due to broadcast communication, those messages must also show up somewhere in the path from p to r, however. The copy of m dedicated to receiver r will hence see at least some of those messages also ahead of it. In other words, this message cannot take on the smallest possible delay value in this case, as it does not arrive in an "empty" system. Hence, the smallest delays must be larger than τ^-, at least for some messages.

4 Clock Synchronization in the Θ-Model

In [13, 12], we introduced and analyzed a clock synchronization algorithm that can be employed in the Θ-Model which will be the core of the novel failure detector algorithm of Section 5. It assumes that every process p is equipped with an adjustable integer-valued clock $C_p(t)$, which can be read at arbitrary real-times t. The clock synchronization algorithm at p is in charge of maintaining $C_p(t)$, in a way that guarantees the following system-wide properties:

(P) [illegible] There is some constant [illegible] $D_{max} > 0$ such that

$$|C_p(t) - C_q(t)| \leq D_{max} \tag{1}$$

for any two processes p and q that are correct up to real-time t.

(A) [illegible] There are some constants $R^-, O^-, R^+, O^+ > 0$ such that

$$O^-(t_2 - t_1) - R^- \leq C_p(t_2) - C_p(t_1) \leq O^+(t_2 - t_1) + R^+ \tag{2}$$

for any process p that is correct up to real-time $t_2 \geq t_1$.

Informally, (P) states that the difference of any two correct clocks in the system must be bounded, whereas (A) guarantees some relation of the progress of clock time with respect to the progress of real-time; (A) is also called [illegible] in the literature. Note that (P) and (A) are [illegible] [24], in the sense that they hold also for processes that crash or become otherwise faulty later on.

Figure 2 shows a simplified version of the clock synchronization algorithm in [13].[4] Based upon the number of processes that have completed booting, two modes of operation must be distinguished here: In [illegible], less than $n-f$ correct processes are up and running. Our algorithm maintains both precision (P) and the upper envelope bound (i.e., fastest progress) in (A) here for all processes. Soon after the $n-f$-th correct process gets up, the system makes the transition to [illegible], where it also guarantees the lower envelope (i.e., slowest progress). This holds for all processes that got synchronized (termed [illegible] in [13]) by executing `line 19` in Figure 2 at least once.

In this section, we review some results of our detailed analysis in [13], as far as they are required for this paper. We start with some useful definitions.

[4] Additionally to the algorithm, [13] presents a full analysis of the algorithm under the perception-based hybrid failure model of [25].

```
VAR k : integer := 0;

/* Initialization */
send (init, 0) to all [once];

if received (init, 0) from process $p$
   → if an (echo) was already sent
         → re-send last (echo) to $p$
      else → re-send (init, 0) to $p$
      fi
fi

if received (init, k) from at least $f + 1$ distinct processes
   → send (echo, k) to all [once];
fi

if received (echo, k) or (echo, $k + 1$) from at least $f + 1$ distinct
   processes
   → send (echo, k) to all [once];
fi
if received (echo, k) or (echo, $k + 1$) from at least $n - f$ distinct
   processes
   → $C := k + 1$; /* update clock */
      $k := k + 1$;
      send (init, k) to all [once]; /* start next round */
fi
if received (echo, $\ell$) or (echo, $\ell + 1$) from at least $f + 1$ distinct
   processes with $\ell > k$
   → $C := \ell$; /* update clock */
      $k := \ell$; /* jump to new round */
      send (echo, k) to all [once];
fi
```

Fig. 2. Clock Synchronization Algorithm

Definition 1 (Local Clock Value). $C_p(t)$ [illegible] p [illegible] t. σ_p^k [illegible] $k \geq 0$ [illegible] p [illegible] $k + 1$

Definition 2 (Maximum Local Clock Value). $C_{max}(t)$ [illegible] t [illegible] $\sigma_{first}^k = \sigma_p^k \leq t$ [illegible] p [illegible] $k + 1 = C_{max}(t)$

We proceed with the properties that can be guaranteed during both degraded mode and normal mode. The following Lemma 1 gives the maximum rate at which clock values of correct processes could increase.

Lemma 1 (Fastest Progress). [illegible] p [illegible] k [illegible] t [illegible] $k' > k$ [illegible] $t + 2\tau^-(k' - k)$

Theorem 1 specifies the precision D_{MCB} that is eventually achieved by all correct clocks. For processes that boot simultaneously, it holds right from the start. Late starting processes could suffer from a larger precision $D_{max} = \lfloor \Theta + 2 \rfloor$ during a short time interval (duration at most $2\tau^+$) after getting up, but are guaranteed to also reach precision D_{MCB} after that time.

Theorem 1 (Precision). [illegible] $n_{up} \leq n$ [illegible] $n \geq 3f+1$ [illegible] $|C_p(t) - C_q(t)| \leq D_{MCB}$ [illegible] p [illegible] q [illegible] t $D_{MCB} = \lfloor \frac{1}{2}\Theta + \frac{3}{2} \rfloor$

The properties stated so far can be achieved during degraded mode, with any number n_{up} of participating processes. Unfortunately, they do not guarantee progress of clock values. In normal mode, however, the following additional properties can be guaranteed:

Lemma 2 (Slowest Progress). [illegible] p [illegible] k [illegible] t [illegible] $k' > k$ [illegible] $t + 2\tau^+(k' - k)$

Theorem 2 (Simultaneity). [illegible] k [illegible] t [illegible] k [illegible] $t + \tau^+ + \varepsilon$

Theorem 3 (Precision in Normal Mode). [illegible] $|C_p(t) - C_q(t)| \leq D'_{MCB}$ [illegible] p [illegible] q [illegible] t [illegible] $D'_{MCB} = \lfloor \Theta + \frac{1}{2} \rfloor$

Finally, the following Theorem 4 shows that normal mode is entered within bounded time after the $n-f$-th correct process got up.

Theorem 4 (Progress into System). [illegible] t [illegible] $n-f$ [illegible] $n \geq 3f+1$ [illegible] $t + 5\tau^+ + \varepsilon$ [illegible] $\min\{D_{MCB}, D'_{MCB}\}$ [illegible]

5 Perfect Failure Detection in the Θ-Model

In this section, we show how to extend the clock synchronization algorithm of Section 4 in order to obtain a perfect failure detector $\mathcal{P}$. We first recall the properties that must be provided by $\mathcal{P}$ [5]:

(SC) [illegible]: Eventually every process that crashes is permanently suspected by every correct process.

(SA) [illegible]: No process is suspected before it crashes.

```
VAR suspect[$\forall q$] : boolean := false;
VAR saw_max[$\forall q$] : integer := 0;

Execute Clock Synchronization from Figure 2

if received $(init, \ell)$ or $(echo, \ell)$ from $q$
    $\rightarrow$ $saw\_msg[q] := max(\ell, saw\_msg[q])$;
fi

whenever clock $C$ is updated do (after updating)
    $\rightarrow$ $\forall q$ suspect[$q$] := $(C - \Xi) >$ saw_msg[$q$];
```

Fig. 3. Failure Detector Implementation

Our failure detector exploits the fact that correct processes always send their clock values — via ([illegible], k) or ([illegible], k) messages — to all. Due to bounded precision, a correct process p can determine a minimal clock value that it must have seen from every process by some specific time. Consequently, if appropriate messages are missing, p must have crashed. Like the algorithm of [7], our solution needs a priori knowledge of an integer constant Ξ only, which is a function of $\bar{\Theta}$ (see Theorem 5). No a priori knowledge of a bound for τ^+ is required here.

5.1 A Simple Perfect FD Algorithm

In order to properly introduce the details of our clock synchronization-based failure detector, we first ignore system startup: We assume in this subsection that all correct processes are initially up and listening simultaneously, i.e., that they cannot miss each others' messages. Faulty processes may be initially dead or may crash at arbitrary times during operation.

The algorithm given in Figure 3 is a simple extension of the clock synchronization algorithm of Figure 2; note that it could dispose of the [illegible]-protocol (`line 5-10` in Figure 2) since we consider simultaneous booting in this section. The first addition is the vector $saw\_max[\forall q]$ that stores, for every process q, the maximum clock tick k received via ([illegible], k) or ([illegible], k). It is written upon every message reception. Whenever a process updates its clock to k (compare `line 19` and `line 24` in Figure 2), it checks $saw\_max[\forall q]$ to find out which processes failed to send messages for tick $k - \Xi$ at least. All those processes are entered into the vector $suspect[\forall q]$, which is the interface to upper layer programs that use the failure detector module. We will now show that, if Ξ is chosen appropriately, the algorithm given in Figure 3 implements indeed the perfect failure detector.

Theorem 5. [illegible] $\Xi \geq min\left\{\left\lceil \frac{3}{2}\bar{\Theta} + \frac{1}{2}\right\rceil, \left\lceil \bar{\Theta} + \frac{3}{2}\right\rceil\right\}$ [illegible] $\bar{\Theta}$ [illegible] Θ [illegible] $n \geq 3f + 1$ [illegible]

[illegible] We have to show that (SC) and (SA) are satisfied.

For showing (SC), it is sufficient to notice that a process that crashes before it updates its clock to $\ell + 1$ will be suspected by every correct process p when

p reaches clock value $k \geq \ell + \Xi$. Since progress of clock values is guaranteed by Lemma 2, every correct process will eventually reach clock value k (in systems with bounded τ^+ such clock value is reached within bounded time — the exact time bound is derived below in Theorem 6).

To prove (SA), we have to show that Ξ is chosen sufficiently large such that every correct process that reaches a clock value k at time t has already received messages for ticks at least $k - \Xi$ by every correct process. In the worst case setting, a correct process p sets its clock to k at instant $\sigma_p^{k-1} = \sigma_{first}^{k-1}$; hence $k = C_{max}(\sigma_p^{k-1})$. From Lemma 1, it follows that $C_{max}(\sigma_p^{k-1} - \tau^+) \geq k - \lceil \frac{1}{2}\Theta \rceil$. Assuming a maximum precision D_{max}, a bound for the smallest possible clock value of a correct process reads $C_{min}(\sigma_p^{k-1} - \tau^+) \geq C_{max}(\sigma_p^{k-1} - \tau^+) - D_{max} = k - \lceil \frac{1}{2}\Theta \rceil - D_{max}$. Consequently, every correct process must have sent a message for tick $C_{min}(\sigma_p^{k-1} - \tau^+)$ by time $\sigma_p^{k-1} - \tau^+$ that arrives at p by time σ_p^{k-1}. Thus, choosing $\Xi \geq \lceil \frac{1}{2}\Theta \rceil + D_{max}$ is sufficient to ensure that p does not incorrectly suspect any correct process.

Since $\lceil x \rceil + \lceil y \rceil \geq \lceil x + y \rceil$ and $\lceil -x \rceil = -\lfloor x \rfloor$, it follows from setting $x = z + w$ and $y = -w$ that $\lceil z + w \rceil \geq \lceil z \rceil + \lfloor w \rfloor$. By setting $z = \frac{1}{2}\Theta$ and $\lfloor w \rfloor = D_{max}$ we employ $\lceil z + w \rceil$ to choose Ξ. Depending on Θ, precision D'_{MCB} or D_{MCB} is smaller and can be used to calculate Ξ (by replacing D_{max}). Hence, using $D'_{MCB} = \lfloor \Theta + \frac{1}{2} \rfloor$, we get $\Xi \geq \lceil \frac{3}{2}\bar{\Theta} + \frac{1}{2} \rceil$. Performing the same calculation for $D_{MCB} = \lfloor \frac{1}{2}\Theta + \frac{3}{2} \rfloor$, we get $\Xi \geq \lceil \bar{\Theta} + \frac{3}{2} \rceil$. □

To find the worst case detection time of our FD algorithm, we have to determine how long it may take from the time a process p crashed with clock value k until all correct processes reach a clock value of $k + \Xi$ and hence suspect p. In the worst case setting, the process p with maximum clock value crashes immediately after reaching it. All other processes must first catch up to the maximum value, and then make progress for Ξ ticks until correctly suspecting p.

Theorem 6 (Detection Time). [illegible] Ξ [illegible] $(2\Xi + 2)\tau^+ - \tau^-$

Assume the worst case: A process p is crashing at time t_c where $k = C_p(t_c) = C_{max}(t_c)$. By Theorem 2 every correct process must reach clock value k by time $t' = t_c + \tau^+ + \varepsilon$. When a correct process reaches clock value $k + \Xi$ it will suspect p. By Lemma 2 every correct process must reach that clock value by time $t = t' + 2\Xi\tau^+ = t_c + \tau^+ + \varepsilon + 2\Xi\tau^+ = t_c + (2\Xi + 2)\tau^+ - \tau^-$. □

Theorem 6 reveals that the detection time depends upon the actual τ^+, i.e., adapts automatically to the current system load. Nevertheless, following the design immersion principle [1, 26], a bound on the detection time can be computed when our algorithm is immersed in some real system. By conducting a worst-case schedulability analysis of our FD, a bound for τ^+ can be established.

5.2 A Failure Detector Algorithm with Startup

In this subsection, we will add system startup to the picture: All processes are assumed to be initially down here, and correct processes get up one after the other at arbitrary times. Faulty processes either remain down or get up before they finally crash. Note that this setting is stricter than the crash-recovery model of [27], since there are no unstable[5] processes allowed in this paper.

Recalling the semantics of $\mathcal{P}$, the properties of our clock synchronization algorithm suggest two approaches for adding system startup to our FD. First, we noted already in Section 4 that our algorithm maintains some precision $D_{max} > D_{MCB}$ during the whole system lifetime. Hence, if we based Ξ upon $D_{max} = \lfloor \Theta + 2 \rfloor$, most of the proof of Theorem 5 would apply also during system startup: (SC) is guaranteed, since progress of clock values is eventually guaranteed, namely, when normal mode is entered. The major part of the proof of (SA) is also valid, provided that D_{MCB} is replaced by D_{max}.

There is one remaining problem with this approach, however: During system startup, the resulting algorithm could suspect a correct process that simply had not started yet. When this process eventually starts, it is of course removed from the list of suspects — but this must not happen in case of the perfect failure detector. Note that not suspecting processes that never sent any message until transition to normal mode does not work either, since a process cannot reliably detect when the transition to normal mode happens. Consequently, unless the perfect FD specification is extended by the notion of "not yet started" processes, there is no hope of implementing $\mathcal{P}$ also during system startup.

The alternative is to accept degraded failure detection properties during system startup: We will show below that the failure detector of Figure 3 implements actually an [illegible] failure detector $\diamond\mathcal{P}$. This FD is weaker than $\mathcal{P}$, since it assumes that there is some time t after which (SA) must hold. In our case, t is the time when the last correct process has completed booting and normal mode is entered. Nevertheless, viewed over the whole system lifetime, our FD algorithm only provides eventual semantics.

Theorem 7 (Eventually Perfect FD). [illegible] $\diamond\mathcal{P}$

[illegible] We have to show that (SC) and (SA) are eventually satisfied. Let t_{up} be the time when the last correct process gets up. Theorem 4 shows that progress of clock values comes into the system and all correct processes are within precision $\min\{D_{MCB}, D'_{MCB}\}$ by time $t_{up} + 5\tau^+ + \varepsilon$. Hence, after that time, the proof of Theorem 5 applies literally and reveals that our algorithm implements $\mathcal{P}$ and hence belongs to the class $\diamond\mathcal{P}$ during the whole system lifetime. □

Theorem 7 reveals that any consensus algorithm that uses $\diamond\mathcal{P}$ under the generalized partially synchronous system model of [5] solves the booting problem if used in conjunction with our FD implementation. After all, the model of [5]

[5] Unstable processes change between up and down infinitely often.

allows arbitrary message losses to occur until the (unknown) global stabilization time GST. $\diamond\mathcal{P}$-based consensus algorithms that work with eventually reliable links can also be used immediately. Such solutions are useful even in the context of real-time systems, since we can bound the time until $\diamond\mathcal{P}$ becomes $\mathcal{P}$. Even if the consensus algorithm is designed to work with $\diamond\mathcal{P}$ it is possible to give termination times when the FD in fact provides the semantics of $\mathcal{P}$.

6 Discussion

It has been taken for granted for many years that fault-tolerant distributed real-time computing problems admit solutions designed in synchronous computational models only. Unfortunately, given the difficulty of ensuring that stipulated bounds on computation times and transmission delays are always met (which is notoriously difficult with many systems, especially those built out of COTS products), the safety/liveness/timeliness properties achieved with such systems may have a poor coverage. This holds true for any design that rests upon (some) timed semantics, including timed asynchronous systems [28] and the Timely Computing Base [29]. With such solutions, the core questions are: How do you set your timers? How do you know your response times?

Some safety properties (e.g. agreement in consensus), and liveness properties, can be guaranteed in purely asynchronous computational models, however. Since asynchronous algorithms do not depend upon timing assumptions, those properties hold regardless of the underlying system's actual timing conditions. The coverage of such time-free solutions is hence necessarily higher than that of a solution involving some timing assumptions. The apparent contradiction between time-free algorithms and timeliness properties can be resolved by following the [illegible] principle, which was introduced in [26] and referred to as the [illegible] principle in [1]. Design immersion permits to consider time-free algorithms for implementing system or application level services in real-time systems, by enforcing that timing-related conditions (like "delay for time X") are expressed as time-free logical conditions (like "delay for x round-trips" or "delay for x events"). Safety and liveness properties can hence be proved [illegible] of the timing properties of the system where the time-free algorithm will eventually be run. Timeliness properties are established only late in the design process, by conducting a worst-case schedulability analysis, when a time-free solution is immersed in a real system with its specific low-level timing properties.

Design immersion can of course be applied to FD-based consensus algorithms, which are purely asynchronous algorithms, and to our partially synchronous time-free FD algorithms (this paper, as well as [7]). Immersion of the Θ-Model into some synchronous model permits to conduct schedulability analyses strictly identical to the analysis given in [1]. The FD algorithm presented in this paper surpasses the one from [7] with respect to detection time: The former algorithm's detection time is bounded by $(3\Theta+3)\tau^+-\tau^-$, whereas our clock synchronization-based solution achieves $(2\Theta+5)\tau^+-\tau^-$. The latter is hence at least as good as the former one, and even better when $\Theta > 2$. Note that the detection time can hence be bounded a priori if a priori bounds $\bar{\tau}^+$, $\bar{\tau}^-$ and $\bar{\Theta}$ are available.

In sharp contrast to the solution of [7], our new FD algorithm properly handles system startup as well, without requiring undue additional assumptions or an increased number of processes. Due to the uncertainty introduced by initially down correct processes, however, it can only provide the semantics of $\diamond\mathcal{P}$. In conjunction with a $\diamond\mathcal{P}$-based consensus algorithm that can handle eventually reliable links, our FD hence allows to solve consensus even during system booting. Since we can bound the time until $\diamond\mathcal{P}$ becomes $\mathcal{P}$, this solution can even be employed in real-time systems.

As said before, our failure detector has advantageous properties regarding assumption coverage. Compared to solutions where timeouts are increased during periods of high system/network load, our approach has the advantage that during overload no increase of timeout values has to be effected. This is due to the fact that, in real systems, there is typically some sufficient correlation between τ^+ and τ^- such that Θ is always maintained. Note that it suffices to have this correlation property holding at least once in a system's lifetime for making the coverage of the Θ-Model greater than the coverage of a non time-free model. (See [10] for a more detailed discussion on the Θ assumption.)

It follows that our failure detector provides the required (time-free) properties (SC) and (SA) while just the detection time possibly increases. Note carefully that the detection time is always as good as provided by the underlying system/network, i.e., the FD timing properties "emerge" naturally from the system/network capabilities [1, 26]. Moreover, if the system/network load returns to expected behavior, our algorithm is still as exact and fast as predicted, while algorithms that adapt their timeouts would have larger detection latencies then.

Since failure detection is often considered as a system service provided to application-level algorithms, the overhead of the failure detector implementation is an important issue. It might seem that this paper's algorithm induces an excessive overhead as there are always FD-level messages in transit. This is not true, however, since one must distinguish between message delay (in time units) and the throughput of a link (in messages per time unit).

A logical link in real distributed systems consists of various outbound message queues, the physical link and the inbound message queues. If a message m is in transit for δ real-time units, obviously not all resources belonging to a logical links are used by m during the whole interval δ but only at most one at a time (assuming m is in some queue, waiting for being chosen by the scheduler to be processed next, one could say no resource is allocated to m at this time); thus the overhead is never 100%. In systems with a reasonably large delay$\times$bandwidth product, the overhead is in fact quite small.

Consider the extreme case of a satellite broadcast communication link, for example, where the end-to-end propagation delay typically is in the order of 300 ms. With up to $n = 10$ processors, 1,000 bit long FD messages, and a link throughput of 2 megabit/second, the link occupancy time for this paper's algorithm would be 5 ms per round, entailing a communication overhead smaller than 2%. Moreover, the overhead can be reduced further by introducing local pauses between rounds (which does not even need timers as it can be done by counting suitable local events), see [2] for details.

References

1. Hermant, J.F., Le Lann, G.: Fast asynchronous uniform consensus in real-time distributed systems. IEEE Transactions on Computers **51** (2002) 931–944
2. Le Lann, G., Schmid, U.: How to maximize computing systems coverage. Technical Report 183/1-128, Department of Automation, Technische Universität Wien (2003)
3. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty processor. Journal of the ACM **32** (1985) 374–382
4. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. Journal of the ACM **34** (1987) 77–97
5. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43** (1996) 225–267
6. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. Journal of the ACM **43** (1996) 685–722
7. Le Lann, G., Schmid, U.: How to implement a timer-free perfect failure detector in partially synchronous systems. Technical Report 183/1-127, Department of Automation, Technische Universität Wien (2003)
8. Larrea, M., Fernandez, A., Arevalo, S.: On the implementation of unreliable failure detectors in partially synchronous systems. IEEE Transactions on Computers **53** (2004) 815–828
9. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35** (1988) 288–323
10. Widder, J.: Distributed Computing in the Presence of Bounded Asynchrony. PhD thesis, Vienna University of Technology, Fakultät für Informatik (2004)
11. Larrea, M., Fernández, A., Arévalo, S.: On the impossibility of implementing perpetual failure detectors in partially synchronous systems. In: Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP'02), Gran Canaria Island, Spain (2002)
12. Widder, J.: Booting clock synchronization in partially synchronous systems. In: Proceedings of the 17th International Symposium on Distributed Computing (DISC'03). Volume 2848 of LNCS., Sorrento, Italy, Springer Verlag (2003) 121–135
13. Widder, J., Schmid, U.: Booting clock synchronization in partially synchronous systems with hybrid node and link failures. Technical Report 183/1-126, Department of Automation, Technische Universität Wien (2003) (submitted for publication).
14. Srikanth, T.K., Toueg, S.: Optimal clock synchronization. Journal of the ACM **34** (1987) 626–645
15. Dolev, D., Friedman, R., Keidar, I., Malkhi, D.: Failure detectors in omission failure environments. In: Proc. 16th ACM Symposium on Principles of Distributed Computing, Santa Barbara, California (1997) 286
16. Malkhi, D., Reiter, M.: Unreliable intrusion detection in distributed computations. In: Proceedings of the 10th Computer Security Foundations Workshop (CSFW97), Rockport, MA, USA (1997) 116–124
17. Kihlstrom, K.P., Moser, L.E., Melliar-Smith, P.M.: Solving consensus in a byzantine environment using an unreliable fault detector. In: Proceedings of the International Conference on Principles of Distributed Systems (OPODIS), Chantilly, France (1997) 61–75

18. Doudou, A., Garbinato, B., Guerraoui, R., Schiper, A.: Muteness failure detectors: Specification and implementation. In: Proceedings 3rd European Dependable Computing Conference (EDCC-3). Volume 1667 of LNCS 1667., Prague, Czech Republic, Springer (1999) 71–87
19. Doudou, A., Garbinato, B., Guerraoui, R.: Encapsulating failure detection: From crash to byzantine failures. In: Reliable Software Technologies - Ada-Europe 2002. LNCS 2361, Vienna, Austria, Springer (2002) 24–50
20. Basu, A., Charron-Bost, B., Toueg, S.: Simulating reliable links with unreliable links in the presence of process crashes. In Babaoglu, Ö., ed.: Distributed algorithms. Volume 1151 of Lecture Notes in Computer Science. (1996) 105–122
21. Liu, J.W.S.: Real-Time Systems. Prentice Hall (2000)
22. Stankovic, J.A., Spuri, M., Ramamritham, K., Buttazzo, G.C.: Deadline Scheduling for Real-Time Systems. Kluwer Academic Publishers (1998)
23. Albeseder, D.: Experimentelle Verifikation von Synchronitätsannahmen für Computernetzwerke. Diplomarbeit, Embedded Computing Systems Group, Technische Universität Wien (2004) (in German).
24. Hadzilacos, V., Toueg, S.: Fault-tolerant broadcasts and related problems. In Mullender, S., ed.: Distributed Systems. 2nd edn. Addison-Wesley (1993) 97–145
25. Schmid, U., Fetzer, C.: Randomized asynchronous consensus with imperfect communications. In: 22nd Symposium on Reliable Distributed Systems (SRDS'03), Florence, Italy (2003) 361–370
26. Le Lann, G.: On real-time and non real-time distributed computing. In: Proceedings 9th International Workshop on Distributed Algorithms (WDAG'95). Volume 972 of Lecture Notes in Computer Science., Le Mont-Saint-Michel, France, Springer (1995) 51–70
27. Aguilera, M.K., Chen, W., Toueg, S.: Failure detection and consensus in the crash-recovery model. Distributed Computing **13** (2000) 99–125
28. Cristian, F., Fetzer, C.: The timed asynchronous distributed system model. IEEE Transactions on Parallel and Distributed Systems **10** (1999) 642–657
29. Veríssimo, P., Casimiro, A., Fetzer, C.: The timely computing base: Timely actions in the presence of uncertain timeliness. In: Proceedings IEEE International Conference on Dependable Systems and Networks (DSN'01 / FTCS'30), New York City, USA (2000) 533–542

Total Order Communications: A Practical Analysis*

Roberto Baldoni, Stefano Cimmino, and Carlo Marchetti

Dipartimento di Informatica e Sistemistica,
Università di Roma "La Sapienza",
Via Salaria 113, 00198, Roma, Italy
{baldoni, cimmino, marchet}@dis.uniroma1.it

Abstract. Total Order (TO) broadcast is a widely used communication abstraction that has been deeply investigated during the last decade. As such, the amount of relevant works may leave practitioners wondering how to select the TO implementation that best fits the requirements of their applications. Different implementations are indeed available, each providing distinct safety guarantees and performance. These aspects must be considered together in order to build a correct and sufficiently performing application. To this end, this paper analyzes six TO implementations embedded in three freely-distributed group communication systems, namely Ensemble, Spread and JavaGroups. Implementations are first classified according to the enforced specifications, which is given using a framework for specification tailored to total order communications. Then, implementations are compared under the performance viewpoint in a simple yet meaningful deployment scenario. In our opinion, this structured information should assist practitioners (i) in deeply understanding the ways in which implementations may differ (specifications, performance) and (ii) in quickly relating a set of total order algorithms to their specifications, implementations and performance.

1 Introduction

Total Order (TO) is a widely investigated communication abstraction implemented in several distributed systems. Intuitively, a TO primitive ensures that processes of a message-passing distributed system deliver the same sequence of messages. This property is extremely useful for implementing several applications, e.g active software replication [1].

However, there are several subtleties that still deserve clarification, especially among practitioners that can get confused by the relevant amount of work done in this area. A first issue is to understand the guarantees of a TO primitive, as distinct primitives and implementations enforce distinct specifications that

* This work is partially supported by the european project EUPubli.com funded by the European Community and by the italian project MAIS funded by the Italian Ministry of Research.

have to be matched against application correctness requirements. To achieve this, in this paper we first present six existing TO specifications organized into a hierarchy, and then we identify how specifications differ in terms of the possible behavior of faulty processes. Then, we classify into the hierarchy both fixed sequencer and privilege-based TO protocols given in the context of primary component group communications [2, 3], by also pointing out real systems implementing these primitives. These are the results of a formal analysis available in a companion paper [4].

A further issue we deem relevant for practitioners is performance. Several works present performance analysis of TO primitives, e.g. [5, 6]. Some other works discuss the correlation between the guarantees and the achievable performance of a TO implementation, e.g. [7]. These works mainly focus on intrinsic characteristics of the analyzed primitives, and not on the overall system in which a primitive is typically implemented. Therefore in this paper, in order to assist practitioners in finding the TO implementation that best matches both applications' correctness and performance requirements, we present a simple yet meaningful performance analysis of the implementations in real systems of the discussed TO primitives.The results show that the performance of a TO primitive depends on the combination of three factors, (i) the enforced TO specification, (ii) the TO protocol used to implement that specification, and (iii) the way the protocol is implemented.

The remainder of this paper is organized as follows. Section 2 introduces total order broadcast. In particular, it describes the system model, the properties defining the TO problem, a hierarchy of TO specifications, and highlights their differences in terms of the admitted behavior of faulty processes. Then, Section 3 presents fixed-sequencer and privilege-based TO implementations provided by group communication systems. Section 4 describes some real systems implementing TO primitives and compares them from a performance point of view (Appendix A gives further details about the configuration of these systems). Finally, Section 5 concludes the paper.

2 Total Order Broadcast

2.1 System Model

[illegible] We consider a system composed by a finite set of processes $\Pi = \{p_1 \dots p_n\}$ communicating by message passing. Each process behaves according to its specification until it possibly crashes. A process that never crashes is [illegible], while a process that eventually crashes is [illegible]. The system is asynchronous, i.e. there is no bound known or unknown on message transfer delays and on processes' relative speeds. In order to broadcast a message m, a process invokes the $\mathsf{TOcast}(m)$ primitive. Upon receiving a message m, the underlying layer of a process invokes the $\mathsf{TOdeliver}(m)$ primitive, which is an upcall used to deliver m to the process. We say that a process $p \in \Pi$ [illegible] a message m [illegible] it executes $\mathsf{TOcast}(m)$. Analogously, we say that a process $p \in \Pi$ todelivers a message m [illegible] it executes $\mathsf{TOdeliver}(m)$.

[illegible] Each process $p \in \Pi$ can experience the occurrence of three kinds of events, namely $TOcast(m)$, $TOdeliver(m)$ and *crash*. An history h_p is the sequence of events occurred at p during its lifetime. A [illegible] is a set of histories h_{p_i}, one for each process $p_i \in \Pi$. Informally speaking, a [illegible] P is a predicate defining a set R_P of system runs, composed by all system runs whose process histories satisfy P. A specification, denoted $S(P_1 \dots P_m)$ (with $m \geq 1$) is the conjunction of m properties, thus defining a set R_S of system runs, composed by those runs satisfying all properties in S. Given two specifications $S(P_1 \dots P_m)$ and $S'(P'_1 \dots P'_\ell)$, we say that S is stronger than S', denoted $S \rightarrow S'$, [illegible] $R_S \subset R_{S'}$. In this case we also say that S' is weaker than S. Finally, two specifications S and S' are said to be equivalent, denoted $S \equiv S'$, [illegible] $R_S \equiv R_{S'}$.

2.2 Total Order Properties

Total order broadcast is specified by means of four properties, namely [illegible], [illegible], [illegible], and [illegible]. Informally speaking, a [illegible] property guarantees that messages sent by correct processes are eventually delivered at least by correct processes; an [illegible] property guarantees that no spurious or duplicate messages are delivered; an [illegible] property ensures that (at least correct) processes deliver the same set of messages; an [illegible] property constrains (at least correct) processes delivering the same messages to deliver them in the same order. Each property can be formally defined in distinct ways, thus generating distinct specifications. As an example, properties can be defined as [illegible] or [illegible], being non-uniform ones less restrictive, as they allow arbitrary behavior for faulty processes.[1]

[illegible] properties can be further distinguished on the basis of the possibility to have gaps in the sequence of messages delivered by processes, and are thus classified into [illegible] and [illegible] properties. A weak [illegible] property requires a pair of processes delivering the same pair of messages to deliver them in the same order. This restriction does not prevent a process p to skip the delivery of some messages. Therefore, it allows the occurrence of gaps in the sequence of messages delivered by p with respect to those delivered by other processes. In contrast, a strong [illegible] property avoids gaps in the sequence of delivered messages as it requires that two processes delivering a message m have delivered exactly the same ordered sequence of messages before delivering m.

Table 1 reports the definition of each property. In particular, we consider both uniform and non-uniform formulations for [illegible], i.e. [illegible] (UA) and [illegible] (NUA), and the four [illegible] properties arising

[1] It is worth noting that uniform properties are meaningful only in certain environments. For instance, uniform properties are not enforceable assuming malicious fault models.

Table 1. Definition of the properties defining TO specifications

	Validity and Integrity properties	
NUV	$\triangleq$	If a *correct* process tocasts a message m, then it eventually todelivers m
UI	$\triangleq$	For any message m, every process p todelivers m at most once, and only if m was previously tocast by some process
	Agreement properties	
UA	$\triangleq$	If a process todelivers a message m, then all correct processes eventually todeliver m
NUA	$\triangleq$	If a *correct* process todelivers a message m, then all correct processes eventually todeliver m
	Order properties	
SUTO	$\triangleq$	If some process todelivers message m before message m', then a process todelivers m' only after it has todelivered m
SNUTO	$\triangleq$	If some *correct* process todelivers message m before message m', then a *correct* process todelivers m' only after it has todelivered m
WUTO	$\triangleq$	If processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'
WNUTO	$\triangleq$	If *correct* processes p and q both todeliver messages m and m', then p todelivers m before m' if and only if q todelivers m before m'

from the combination of uniform and non-uniform with strong and weak formulations, i.e. Strong Uniform Total Order ($SUTO$), Strong Non-uniform Total Order ($SNUTO$), Weak Uniform Total Order ($WUTO$) and Weak Non-uniform Total Order ($WNUTO$). Finally, we consider [illegible] (NUV) and [illegible] (UI), as the latter can be easily implemented, thus appearing in almost all TO specifications, while the former is the only [illegible] property meaningful in our system model (i.e. [illegible] cannot be implemented). Interested readers can refer to [4] for deeper explanations of differences and relations among these properties.

2.3 A Hierarchy of Total Order Specifications

Assuming NUV and UI, it is possible to combine [illegible] and [illegible] properties to obtain six significant TO specifications. We denote $TO(A, O)$ the TO specification $S(NUV, UI, A, O)$, where $A \in \{UA, NUA\}$ and $O \in \{SUTO, WUTO, WNUTO\}$.[2]

It is possible to identify several $\rightarrow$ relations among these TO specifications [4]. Figure 1 shows that these specifications represent a hierarchy by depicting the transitive reduction of the $\rightarrow$ relation among TO specifications.

Let us note that the root of the hierarchy, i.e. $TO(UA, SUTO)$, is the specification closest to the intuitive notion of total order broadcast, as it imposes that [illegible]. In contrast, weaker specifications admit runs in which faulty processes may exhibit a larger set of behaviors, as discussed in the following section. It is worth noting that weaker specifications are implemented in several real systems, e.g. Ensemble [8], JavaGroups [9].

[2] In [4] we show that $TO(NUA, SNUTO) \equiv TO(NUA, WNUTO)$ and that $TO(UA, SNUTO) \equiv TO(UA, WNUTO)$.

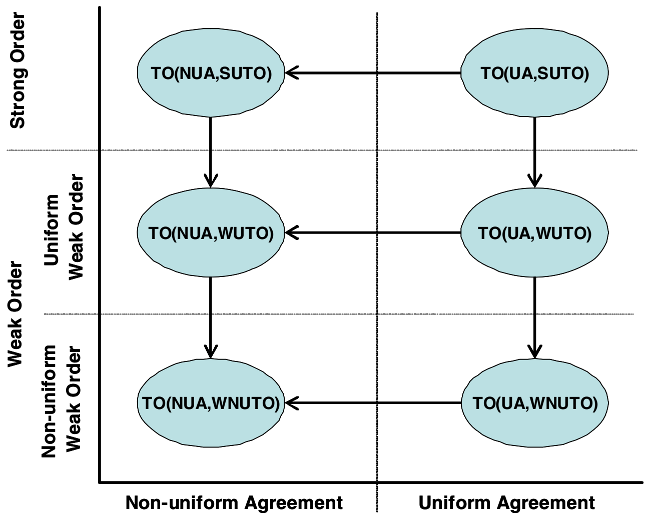


Fig. 1. A hierarchy of TO specifications

2.4 On the Behavior of Faulty Processes

Each TO specification constrains all correct processes to deliver exactly the same ordered set of messages. Differences among the sequences of messages delivered by faulty and correct processes can be characterized using the following patterns.

EP1: a faulty process p delivers a prefix of the ordered set of messages delivered by correct processes;

EP2: a faulty process p delivers some messages not delivered by correct processes;

EP3: a faulty process p skips the delivery of some messages delivered by correct processes;

EP4: a faulty process p delivers some messages in an order different from correct processes.

Table 2. Possible differences between the behavior of faulty and correct processes

TO specification	Admitted execution patterns
$TO(UA, SUTO)$	EP1
$TO(UA, WUTO)$	EP1 or EP3
$TO(UA, WNUTO)$	EP1 or EP3 or EP4
$TO(NUA, SUTO)$	EP1 or EP2
$TO(NUA, WUTO)$	EP1 or EP2 or EP3
$TO(NUA, WNUTO)$	EP1 or EP2 or EP3 or EP4

Each specification allows the occurrence of one or more of the above execution patterns. Moreover, from the definition of the $\rightarrow$ relation, it follows that for each pair of specifications $S, S' : S \rightarrow S'$, S' allows at least all execution patterns admitted by S. For example, $TO(UA, SUTO)$ allows EP1 while $TO(UA, WUTO)$ allows EP1 and EP3. Table 2 shows for each specification the admitted execution patterns. Let us note that these execution patterns are formally derived from specifications [4].

3 TO Implementations in Group Communication Systems

Group communication systems are one of the most successful class of systems implementing TO primitives. These systems adopt several distinct architectures [10]. For the sake of clarity, in the remainder of this paper we use a simplified architecture depicted in Figure 2, in which a Total Order layer implements a TO specification by relying on another layer, namely VSC, which provides virtually synchronous communications [11].[3] According to the virtual synchrony programming model, processes are organized into groups. Groups are [illegible], i.e. processes are allowed to join and voluntarily leave a group using appropriate primitives. Furthermore, faulty processes are excluded by groups after crashing. A [illegible] provides each process of a group with a consistent [illegible] v_i composed by the identifiers of all non-crashed processes currently belonging to the group. Upon a membership change, processes agree on a new view through a [illegible]. At the end of this protocol, group members are provided with a view v_{i+1} that (i) is delivered to all the members of v_{i+1} through a view change event, and (ii) contains the identifier of all the members that deliver v_{i+1}. We consider a [illegible] membership service, e.g. [13], guaranteeing that all members of the same group observe the same sequence of views as long as they stay in the group. In this context, the VSC layer guarantees (i) that membership changes of a group occur in the same order in all the members that stay within the group, and (ii) that membership changes are totally ordered with respect to all messages sent by members. It is worth noting that the primary component membership service is not implementable in a non-blocking manner in asynchronous systems [14].[4]

The VSC layer also provides basic communication services. We consider two primitives, namely *Rcast* and *URcast*, which resembles non-uniform and uniform

[3] Let us remark that other approaches incorporating the implementation of *Order* and *Agreement* properties into a single protocol are possible, e.g. [12].

[4] In *partitionable* systems, groups may partition into subgroups (or *components*), e.g. due to network failures, and members of distinct subgroups can deliver distinct sequences of views. In this setting, specifying a total order primitive can drive to complex specifications, e.g. [2], whose usefulness has still to be verified [15]. However, non-blocking implementations of partitionable group membership services are feasible in asynchronous systems.

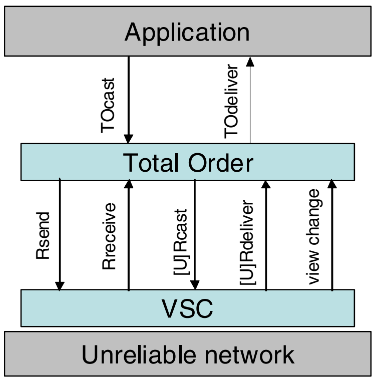


Fig. 2. Reference architecture of a group communication system

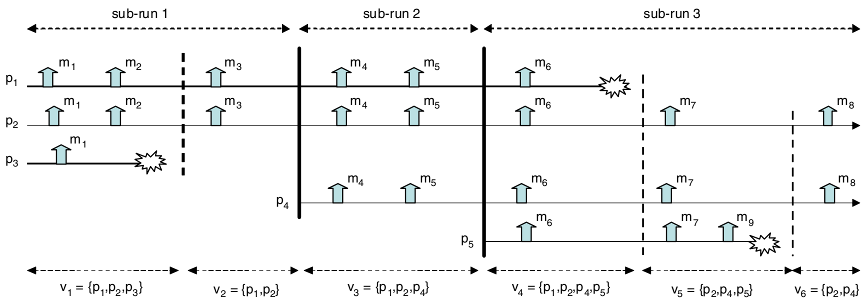


Fig. 3. A run of a system supporting dynamic process joins

reliable broadcast in the context of dynamic groups, respectively. These primitives ensure agreement of message deliveries for processes belonging to a view v_i and installing v_{i+1}, thus enforcing virtual synchrony. *URcast* also prevent faulty processes to deliver messages that will not be delivered by correct processes (i.e., it prevents the occurrence of EP2, in a way similar to *UA*). Interested readers can refer to [4] for a formal definition of these primitives.

3.1 Static Versus Dynamic Group Communications

Following the model proposed by Hadzilacos and Toueg in [16], the properties introduced in Section 2.2 are based on a system model that does not take process joins into account. We now show how the TO specifications introduced in Section 2.3 can be used to classify also dynamic TO implementations, as the one given in the context of group communications. To this aim, we introduce the notion of [illegible], i.e. a portion of the overall computation of a system supporting dynamic groups in which join events may [illegible] appear at the beginning of the sub-run. Consider the computation depicted in Figure 3: it can be decomposed in three static sub-runs, namely sr_1, sr_2, sr_3. A sub-run can be described with events and process histories as those introduced in Section 2.1, i.e. $TOcast(m)$, $TOdeliver(m)$, and *crash*. As an example the sub-run sr_2 depicted in Figure 3 is composed by the histories of processes p_1,p_2 and p_4 containing the message delivery events of m_4 and m_5. Moreover, p_1 is correct in sr_1 and sr_2 while it is faulty in sr_3.

In this dynamic context, the TO specification enforced by a TO implementation $\mathcal{I}$ can be defined as follows.

Definition 1. Let $\mathcal{I}$ be a TO implementation and let $R_{\mathcal{I}}$ be the set of [illegible] that $\mathcal{I}$ can generate. $\mathcal{I}$ enforces a TO specification S [illegible] :
1. $R_{\mathcal{I}} \subseteq R_S$, and
2. $\forall S'\ S' \rightarrow S \Rightarrow R_{\mathcal{I}} \nsubseteq R_{S'}$.

Therefore the problem of finding the TO specification enforced by a TO implementation $\mathcal{I}$ boils down to find a TO specification S defining the smallest superset of $R_{\mathcal{I}}$.

As an example, in the run depicted in Figure 3, sub-runs sr_1 and sr_2 satisfy $TO(UA, SUTO)$, while sr_3 only satisfies $TO(NUA, SUTO)$ (due to p_5 delivering m_9). Therefore, an implementation $\mathcal{I}$ that may generate this run enforces at most $TO(NUA, SUTO)$ (i.e. $\mathcal{I}$ does not enforce $TO(UA, SUTO)$).

3.2 TO Protocols

In this section we analyze the implementation of TO primitives offered by group communication systems. The most widely used protocols implementing the Total Order layer can be classified in [illegible] and [illegible] [7]. Interested readers can refer to [7] for a description of several other classes of TO implementations.

Fixed Sequencer Protocols. In fixed sequencer protocols a particular process, i.e. the [illegible], is responsible for defining message ordering. This process is elected after each view change, usually on the basis of a deterministic rule applied to the current view, and defines a total order of messages by assigning to each message a unique and consecutive sequencer number. The sequence number assigned to a message is sent to all members, which deliver messages according to these numbers. These steps can be implemented using the following communication patterns.

- **Broadcast-Broadcast(BB).** The sender broadcasts message m to all members. Upon receiving m, the sequencer assigns a sequence number *seq* to m and then broadcasts *seq* to all members. As an example, the Ensemble system [8] implements this pattern;
- **Send-Broadcast(SB).** The sender sends message m to the sequencer, which assigns a sequence number *seq* and then broadcasts the pair $\langle m, seq\rangle$ to all members. This pattern is implemented, for example, by the Ensemble system [8];

Table 3. TO specification enforced by each ordering protocol

Ordering protocol	Communication primitive	TO specification
Broadcast-broadcast sequencer	*Rcast*/*Rcast*	$TO(NUA, WNUTO)$
	URcast/*URcast*	$TO(UA, SUTO)$
	Rcast/*URcast*	$TO(NUA, WUTO)$
	URcast/*Rcast*	$TO(UA, WNUTO)$
Send-broadcast sequencer	*Rcast*	$TO(NUA, WNUTO)$
	URcast	$TO(UA, SUTO)$
Ask-broadcast sequencer	*Rcast*	$TO(NUA, WUTO)$
	URcast	$TO(UA, SUTO)$
Privilege-based	*Rcast*	$TO(NUA, WUTO)$
	URcast	$TO(UA, SUTO)$

– **Ask-Broadcast(AB).** The sender first gets a sequence number from the sequencer via a simple rendezvous, then it broadcasts the pair $\langle m, seq\rangle$ to all members. JavaGroups [9] is an example of a system implementing this pattern.

Privilege-Based Protocols. In privilege-based protocols, a single logical token circulates among processes and grants to its holder the privilege to send messages. Each message is sent along with a sequence number derived from a value carried by the token which is increased after each message sent. Receiver processes deliver messages according to their sequence numbers. As only one token may circulate, and only the token holder may send messages, messages are delivered in a total order. Totem [17] and Spread [18] are examples of systems implementing this protocol.

In several privilege-based protocols, e.g. [17, 18, 9], processes are organized in a logical ring, and a process passes the token to the next process upon the occurrence of the first of the following internal events: (i) no more messages to send, or (ii) maximum use of some resources achieved (e.g. maximum token-holding interval, maximum number of messages sent by the process). These kind of protocols usually can be configured to implement *URcast* at the Total Order layer, augmenting *Rcast* with additional mechanisms thanks to the token passing. An example of such protocols is the one implemented by Spread [18].

Table 3 shows the TO specification enforced by each ordering protocol according to Definition 1 given in Section 3.1. In particular, for each protocol, we report the enforced TO specification depending on the used communication primitive, either *Rcast* or *URcast*. Note that BB protocols are based on two broadcasts, which can be performed using different primitives of the VSC layer. The used communication primitives are reported in Table 3 in the form $first\_broadcast/second\_broadcast$. These results have been formally derived in [4], which also includes the pseudo-code of each algorithm.

4 Performance Analysis

Typically, the cost in terms of performance of implementing a property increases with the strenght of the same property. For instance, implementing UA costs more than implementing NUA, as this requires to delay the delivery of messages within processes in order to be sure that they will be delivered by all correct processes. As a consequence, implementations of $TO(NUA, WNUTO)$ are likely to perform better than implementations of other specifications. In the remainder of this section we present a simple performance analysis of some TO implementations of real systems. To this end, we first introduce the group toolkits chosen for evaluation, namely Ensemble [8], Spread [18] and JavaGroups [9]. Then we report the experimental analysis we carried out on such systems.

4.1 Group Communication Toolkits

In this section we exploit the framework defined in the previous sections in order to identify the specifications enforced by TO primitives implemented in the considered group communication systems.

[illegible] Spread is a toolkit designed for large scale networks based on a client-daemon architecture. It offers several communication abstraction, enabled by selecting the so-called “service type”. Spread implements a partitionable membership service based on the [illegible] [19], which extends virtual synchrony to partitionable environments. To comply with the reference architecture of Figure 2, it is thus necessary to assume either absence of network partitioning or the presence of a software filter implementing a primary component membership service and virtual synchrony on top of extended virtual synchrony [19]. In these cases, the privilege-based protocol embedded by Spread (enabled by selecting the `Agreed` service type) implements $TO(NUA, WUTO)$. In contrast, selecting the `Safe` service type, the protocol implements *URcast* on top of *Rcast* (see Section 3.2) and thus the implemented TO specification is $TO(UA, SUTO)$.

[illegible] Ensemble provides fine-grained control over its functionality, which can be selected simply layering [illegible], i.e. well-defined stackable components implementing simple and specific functions. In particular, Ensemble can be configured to implement virtual synchrony and a primary component membership service. A TO primitive is obtained layering a micro-protocol resembling the Total Order layer into a virtually synchronous stack. In the following we consider the micro-protocols named `Seqbb` and `Sequencer`, which correspond to BB and SB fixed sequencer protocols using *Rcast*, respectively (see Section 3.2). As shown in [4], layering one of these protocols in a virtually synchronous stack allows us to enforce $TO(NUA, WNUTO)$.

[illegible] JavaGroups is a Java group communication system based on the concept of micro-protocols (as Ensemble). As for Spread, JavaGroups does not exactly comply with our reference architecture, as it does not provide a primary

Table 4. Main characteristics of the group toolkits with respect to their TO implementations

Toolkit	TO implementation	Protocol type	TO specification
Spread	`Safe`	$PB(URcast)$	$TO(UA, SUTO)$
	`Agreed`	$PB(Rcast)$	$TO(NUA, WUTO)$
Ensemble	`Seqbb`	$BB(Rcast/Rcast)$	$TO(NUA, WNUTO)$
	`Sequencer`	$SB(Rcast)$	$TO(NUA, WNUTO)$
JavaGroups	`TOTAL_TOKEN`	$PB(URcast)$	$TO(UA, SUTO)$
	`TOTAL`	$AB(Rcast)$	$TO(NUA, WUTO)$

component membership service. However, this can be implemented by coding a simple specific micro-protocol [19]. JavaGroups offers two micro-protocols implementing the Total Order layer, namely `TOTAL`, which embeds an AB fixed sequencer protocol using *Rcast*, and `TOTAL_TOKEN`, which embeds a privilege-based protocol enabled to implement *URcast* on top of *Rcast*. As proven in [4], these protocols enforce $TO(NUA, WUTO)$ and $TO(UA, SUTO)$, respectively, if JavaGroups is provided with the primary component membership service micro-protocol.

Tables 3 and 4 summarize the previous discussion. Let us remark that information given by Table 4 hold as long as systems are configured to implement the virtual synchrony model (and not the extended virtual synchrony model), which in some cases requires to extend the toolkit with additional software components, as discussed above (see Appendix A for further details on systems' configurations).

Let us finally note that none of the analyzed toolkit implements all six TO specifications (Ensemble supports other ordering protocols, but they are not able to enforce all remaining specifications).

4.2 Experimental Settings

[illegible] The testbed environment consists of four Intel Pentium 2.5GHz workstations that run Windows 2000 Professional. On each workstation Spread 3.17.0, JavaGroups 2.0.6 and Ensemble 1.40 have been installed and configured. The workstations are interconnected by a 100Mbit Switched Ethernet LAN.

[illegible] All the experiments involve a static group of four processes, each running on a distinct workstation.[5]

We run a distinct experiment for each row of Table 4, in order to evaluate the performance of each TO implementation. Every experiment consists of ten failure-free rounds. During each round, every process sends a burst of B messages

[5] In the case of Spread, which adopts a daemon-based architecture, we run both a daemon and a client on each workstation.

using the TO protocol under examination. Each message has a payload composed by the sender identifier and a local timestamp. The size of the payload is thus very small, i.e. about 8 bytes. After sending the burst, each process waits to deliver all of its messages and those sent by other members (i.e. it waits to deliver $T = 4 \times B$ messages). Each time a process delivers one of the messages sent by itself, it evaluates the message latency exploiting the timestamp contained in the payload. At the end of the experiment, each process evaluates the average message latency. Per-process average message latencies are further averaged to obtain a system message latency. Furthermore, we evaluate the overall system throughput, which is obtained as the sum of the throughput experienced by each process in the experiment. This is in turn calculated as the average number of messages delivered per second during each round. Results were obtained by letting the burst size B vary in $\{1, 10, \dots 100\}$, repeating each experiment 10 times and averaging the results.

We decided to test group toolkits under bursty traffic as developers usually encounter problems in these settings [5].

4.3 Experimental Results

Figure 4 and 5 show the overall comparison. In particular, Figure 4 depicts the average message latency, and Figure 5 presents the overall system throughput as a function of B.

The results can be evaluated under several aspects. In particular, the different behavior of the tested configurations depends on (i) the TO specification implemented by the configuration, (ii) the TO protocol used to implement that specification, and (iii) the way the protocol is implemented, which accounts for different architectures, optimizations, implementation language, etc.

Let us first analyze implementations enforcing the same specifications. Concerning JavaGroups (`TOTAL_TOKEN`) and Spread (`Safe`), which enforce $TO(UA, SUTO)$, this two configurations exploit a similar privilege-based protocol. Spread (`Safe`) outperforms JavaGroups (`TOTAL_TOKEN`), as the average message latency experienced with the latter configuration is about 2 to 3 times the one obtained with the former, while the overall system throughput suffers from a reduction of about 30% to 60%. A similar argument applies to JavaGroups (`TOTAL`) and Spread (`Agreed`), both implementing $TO(NUA, WUTO)$. In this case, the performance gain obtained with Spread's configuration is even more evident. For example, the latency experienced with JavaGroups is about 4 to 6 times the one obtained with Spread. Finally, the two Ensemble configurations, both implementing $TO(NUA, WNUTO)$, perform almost the same.

From the above discussion, it is evident that the performance of a TO primitive enforcing a given TO specification substantially depends on the overall characteristics of the system implementing it. In fact, in spite of implementing the same protocol (and thus the same TO specification), Spread (`Safe`) and JavaGroups (`TOTAL_TOKEN`) gives substantially different performance. This is due to several factors, e.g. implementation language (C++ vs. Java), architectural design and optimizations.

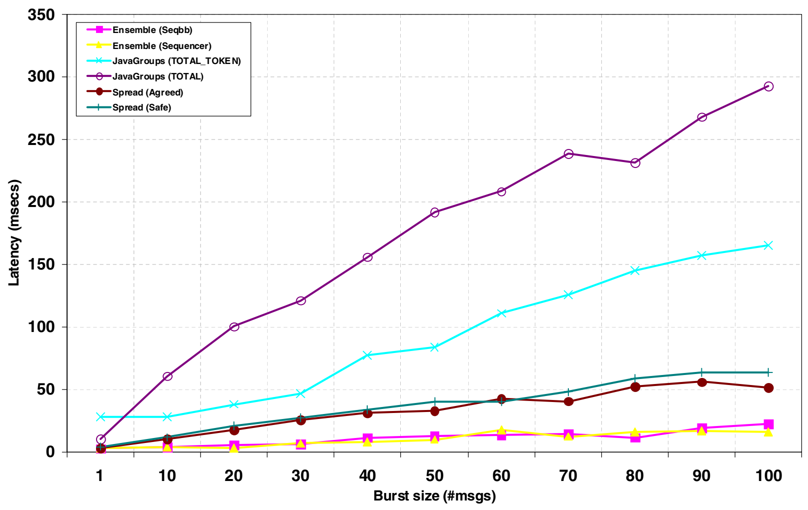


Fig. 4. Average message latency

A second step is therefore to compare different configurations of the same system, in order to avoid biases stemming from implementation issues. Concerning Spread's configurations, `Agreed` outperforms `Safe`. Differences are due to the increased amount of synchronization required by Spread (`Safe`) to enforce $TO(UA, SUTO)$. Very interestingly, the performance penalty paid by Spread (`Safe`) is small, both in terms of additional message latency (1.2 times the one of Spread (`Agreed`)) and in terms of throughput reduction (15% on average). In contrast, JavaGroups (`TOTAL_TOKEN`) outperforms JavaGroups (`TOTAL`), with the latter experiencing twice the average message latency and an average throughput reduction of 30% with respect to the former. These results are unexpected, as JavaGroups (`TOTAL`) implements a TO specification weaker than the one implemented by JavaGroups (`TOTAL_TOKEN`) (i.e. $TO(NUA, WUTO)$ vs. $TO(UA, SUTO)$). We argue that these results are due to JavaGroups (`TOTAL`) embedding an AB fixed sequencer protocol, which is not well suited for configurations in which processes frequently generate bursts of messages of small size. Furthermore, this algorithm suffers from the load to which the sequencer is subject during the experiments. In other words, the experimental settings seem to favor the privilege-based algorithm of JavaGroups (`TOTAL_TOKEN`), which is thus able to perform better, even though implementing a stronger specification.

A final note is on the two Ensemble configurations. In a setting providing hardware broadcast, as the one used for the experiments, BB and SB algorithms perform very similarly. Furthermore, these two configurations exhibit the best results, having the lowest message latency and the highest throughput. In particular, the average message latency experienced with Spread is about 2 to 5 times the one obtained with the two Ensemble configurations. This ratio roughly increases up to 15, if we consider JavaGroups. Concerning throughput, Spread's

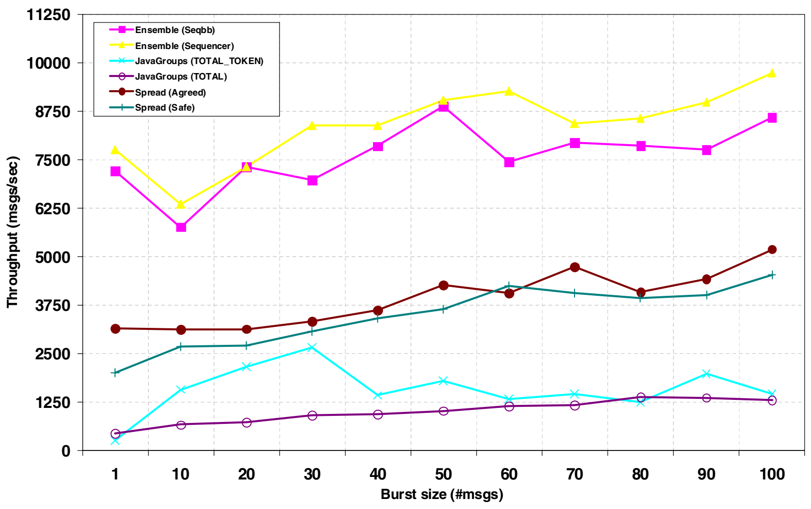


Fig. 5. Overall system throughput

configurations exhibit a reduction of about 50% with respect to Ensemble's configurations. Considering JavaGroups, the throughput reduction is about 80%. These results can be explained noting that Ensemble's configurations implement the weakest TO specifications, i.e. $TO(NUA, WNUTO)$.

4.4 Discussion

The main contributions regarding performance of TO implementations either compare protocols using simulations and/or analytical studies, e.g. [20, 6] or deal with experiments done comparing several algorithms embedded into a single framework, e.g. [5]. We deem that this information does not fully enable immediate comparison of real systems, which is important from a developer's point of view. Upon building an application and thus having to match correctness and performance requirements, a comparison of TO protocols in a simulated environment or in a single real framework is not sufficient, especially in case the developer wishes to select the best TO implementation choosing from a set of available ones. This set of experiments is aimed to complement the available information in order to facilitate this selection. Our plan is to provide a practitioner with a largest set of experiments in the near future, to cope with a larger set of application scenarios including actual failures.

5 Concluding Remarks

This paper provided practitioners with a comprehensive yet quick and easy to understand reference for dealing with total order communications. Existing TO specifications have been classified into an hierarchy which actually models their differences in terms of admitted scenarios. Furthermore, six TO implementations provided by three freely-available systems have been analyzed, matching them against the hierarchy and comparing them from a performance point of view. On the basis of this information complemented with the one provided by simulations [20, 6, 5], practitioners will be able (i) to understand which TO specification meets their application's safety requirements, and (ii) to select the available TO implementation enforcing that TO specification while yielding the best performance. Concerning the latter issue, our experiments point out that the performance of a TO primitive is clearly dependent on the enforced specification, other than the employed algorithm and implementation specific details.

References

1. Schneider, F.B.: Replication Management Using the State Machine Approach. In Mullender, S., ed.: Distributed Systems. ACM Press - Addison Wesley (1993)
2. Chockler, G., Keidar, I., Vitenberg, R.: Group Communications Specifications: a Comprehensive Study. ACM Computing Surveys **33** (2001) 427–469
3. Powell, D.: Group Communication. Communications of the ACM **39** (1996) 50–97
4. Baldoni, R., Cimmino, S., Marchetti, C.: Total order communications over asynchronous distributed systems: Specifications and implementations. Technical Report 06/04, Università di Roma "La sapienza" (2004) Available at http://www.dis.uniroma1.it/∼midlab/publications.html.
5. Friedman, R., van Renesse, R.: Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. Technical Report TR95-1527, Department of Computer Science, Cornell University (1995) Submitted for publication.
6. Dèfago, X.: Agreement-Related Problems: From Semi Passive Replication to Totally Ordered Broadcast. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland (2000) PhD thesis no. 2229.
7. Dèfago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. Technical Report IC/2003/56, École Polytechnique Fédérale de Lausanne, Switzerland (2003)
8. Hayden, M.: The Ensemble System. PhD thesis, Cornell University, Ithaca, NY, USA (1998)
9. Ban, B.: Design and implementation of a reliable group communication toolkit for java. Cornell University (1998)
10. Mena, S., Schiper, A., Wojciechowski, P.: A step towards a new generation of group communication systems. Technical Report IC/2003/01, École Polytechnique Fédérale de Lausanne, Switzerland (2003)
11. Birman, K., Joseph, T.: Exploiting Virtual Synchrony in Distributed Systems. In: Proceedings of the 11th ACM Symp. on Operating Systems Principles. (1987) 123–138
12. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving Consensus. Journal of ACM **43** (1996) 685–722
13. Birman, K., van Renesse, R.: Reliable Distributed Computing with the ISIS toolkit. IEEE CS Press (1994)
14. Chandra, T., Hadzilacos, V., Toueg, S., Charron-Bost, B.: On the Impossibility of Group Membership. In: Proc. of the 15th ACM Symposium of Principles of Distributed Computing. (1996)

15. Schiper, A.: Dynamic group communication. Technical Report IC/2003/27, École Polytechnique Fédérale de Lausanne, Switzerland (2003)
16. Hadzilacos, V., Toueg, S.: Fault-Tolerant Broadcast and Related Problems. In Mullender, S., ed.: Distributed Systems. Addison Wesley (1993)
17. Amir, Y., Moser, L., Melliar-Smith, P.M., Agarwal, D., Ciarfella, P.: The Totem single-ring ordering and membership protocol. ACM Transactions on Computer Systems **13** (1995) 93–132
18. Amir, Y., Stanton, J.: The Spread Wide Area Group Communication System. Technical Report CNDS-98-4, Center for Networking and Distributed Systems, Computer Science Department, Johns Hopkins University, 3400 N. Charles Street Baltimore, MD 21218-2686 (1998)
19. Moser, L.E., Amir, Y., Melliar-Smith, P.M., Agarwal, D.A.: Extended Virtual Synchrony. In: Proceedings of the 14 International Conference on distributed Computing Systems. (1994)
20. Cristian, F., de Beijer, R., Mishra, S.: A Performance Comparison of Asynchronous Atomic Broadcast Protocols. Distributed Systems Engineering Journal **1** (1994) 177–201

Appendix A: Group Toolkit Configuration

This appendix gives additional details on how to configure the systems analyzed in the paper to use their TO primitives (see Table 5). For further details, interested readers are referred to systems' reference manuals.

Table 5. Configurations and additional mechanisms necessary to achieve TO specifications supported by each of the examined group toolkits

Toolkit	Configuration	Additional mechanisms
Spread (Safe)	`Safe` service type	VS + PC GMS filters
Spread (Agreed)	`Agreed` service type	VS + PC GMS filters
Ensemble (BB)	VS + PC GMS + `Seqbb`	-
Ensemble (SB)	VS + PC GMS + `Sequencer`	-
JavaGroups (TB)	VS + `TOTAL_TOKEN`	PC GMS filter
JavaGroups (AB)	VS + `TOTAL`	PC GMS filter

Developers must simply label messages to enact Spread's services. In particular, the `Agreed` label enables the Spread (`Agreed`) configuration, whereas the `Safe` label triggers the Spread `Safe` configuration. There is no need for further configurations. However, developers must provide an implementation of virtual synchrony and primary component membership service filters to achieve the TO specifications described in Table 5.

In Ensemble each process has to specify the stack to use upon joining the group. This can be done either by specifying desired (which identify portions of protocol stacks), or by directly selecting the micro-protocols. In both cases, it is necessary to set a particular field in the data structure representing the so-called join options. In the first case, the `properties` string should be set. The string

```
Gmp:Sync:Heal:Frag:Suspect:Flow:Slander:Total:Primary
```

allows to achieve a TO primitive enforcing $TO(NUA, WNUTO)$ by means of an ordering protocol in a stack also providing virtual synchrony and a primary component membership service. This configuration automatically selects `Seqbb` as the ordering protocol. To use a different protocol, it is necessary to explicitly select all the protocols of the stack. In this case, the `protocol` string should be set, e.g. to

```
Top:Heal:Primary:Present:Leave:Inter:Intra:Elect:Merge:Slander:Sync:Suspect:Stable:Vsync:
Frag_Abv:Partial_appl:Seqbb:Collect:Frag:Pt2ptw:Mflow:Pt2pt:Mnak:Bottom
```

which corresponds to the previous string of properties. To use other TO protocols, it is necessary to substitute `Seqbb` with the protocol to be used, e.g. `Sequencer`, in the string above.

[illegible] Also in JavaGroups the protocols composing the stack can be specified through a string. As an example, the string

```
UDP:PING:FD_SOCK:VERIFY_SUSPECT:STABLE:NACKACK:UNICAST:FRAG:TOTAL_TOKEN:FLUSH:GMS:QUEUE
```

represents a stack providing a total order through the `TOTAL_TOKEN` protocol. Instead, the string

```
UDP:PING:FD_SOCK:VERIFY_SUSPECT:STABLE:NACKACK:UNICAST:FRAG:FLUSH:GMS:TOTAL:QUEUE
```

can be used to exploit the `TOTAL` protocol. However, developers have to implement a primary component membership service filter, which has to be inserted into the stack in order to achieve TO primitives compliant with $TO(UA, SUTO)$ and $TO(NUA, WUTO)$, respectively.

Gracefully Degrading Fair Exchange with Security Modules

Gildas Avoine[1], Felix Gärtner[2], Rachid Guerraoui[3], and Marko Vukolić[3]

[1] Security and Cryptography Laboratory, EPFL, Switzerland
[2] Dependable Distributed Systems Laboratory, RWTH Aachen University, Germany
[3] Distributed Programming Laboratory, EPFL, Switzerland

Abstract. The *fair exchange* problem is key to trading electronic items in systems of mutually untrusted parties. In modern variants of such systems, each party is equipped with a security module. The security modules trust each other but can only communicate by exchanging messages through their untrusted host parties, that could drop those messages.

We describe a synchronous algorithm that ensures deterministic *fair exchange* if a majority of parties are honest, which is optimal in terms of resilience. If there is no honest majority, our algorithm degrades gracefully: it ensures that the probability of unfairness can be made arbitrarily low.

Our algorithm uses, as an underlying building block, an early-stopping subprotocol that solves, in a general omission failure model, a specific variant of consensus we call *biased consensus*. Interestingly, this modular approach combines concepts from both cryptography and distributed computing, to derive new results on the classical fair exchange problem.

1 Introduction

1.1 Motivation

Fair exchange (see e.g. [5, 6, 7, 10, 11, 12, 14, 31]) is a fundamental problem in systems with electronic business transactions. In fair exchange, the participating parties start with an item they want to trade for another item. They possess an executable description of the desired item, typically a boolean function with which an arbitrary item can be checked for the desired properties. Furthermore, they know from which party to expect the desired item and which party is expecting their own item. An algorithm that solves fair exchange must ensure that every honest party eventually either delivers its desired item or aborts the exchange ([illegible] property). The abort option however is excluded if no party misbehaves and all items match their descriptions ([illegible] property). The algorithm should also guarantee that, if the desired item of any party does not match its description, then no party can obtain any (useful) information about any other item ([illegible] property).

Fair exchange is easily solvable using a [illegible] through which all items can be exchanged [13]. The involvement of the trusted third party can be

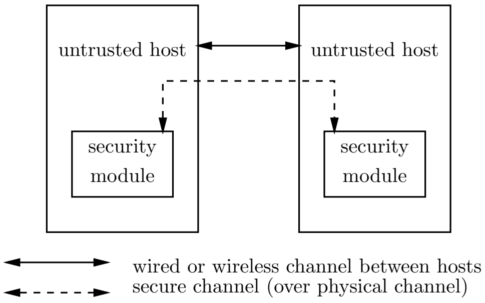

untrusted host
untrusted host
security module
security module
wired or wireless channel between hosts
secure channel (over physical channel)

Fig. 1. Hosts and security modules

reduced using [illegible] schemes where participation of the trusted third party is only necessary if something goes wrong [1]. The context of this paper is one where the trusted third party is a [illegible] entity, distributed within all untrusted parties, as we explain below.

We consider in this paper a system where each party hosts a [illegible] that is [illegible] (Fig. 1). Recently, manufacturers have begun to equip hardware with such modules: these include for instance the "Embedded Security Subsystem" within the recent IBM Thinkpad, or the IBM 4758 secure co-processor board [15]. In fact, a large body of computer and device manufacturers has founded the Trusted Computing Group (TCG) [32] to promote this idea. Besides security modules being tamper proof, the software running within the security modules is certified and they can communicate through secure channels. In certain settings, the overall system can even assumed to be synchronous, i.e., it is reasonable to assume an upper bound on the relative speeds of honest parties (and their security modules) as well as on the communication delays between them. However, dishonest parties can still drop messages exchanged between the underlying security modules in order to violate the fairness of the exchange in their favor, i.e., obtain an item without giving away their own.

The contribution of this paper is a synchronous distributed algorithm aimed at exchanging electronic items among multiple untrusted parties, each hosting a security module. The algorithm provides the two following complementary features:

1. If a majority of parties is honest, then the algorithm deterministically guarantees the termination, effectiveness and fairness properties of fair exchange. This is optimal in terms of resilience: we indeed show that, even in a synchronous model with security modules, no deterministic algorithm solves fair exchange if half of the parties are dishonest.
2. If at least half of the parties turn out to be dishonest, then our algorithm degrades gracefully in the following sense. It still guarantees the termination and effectiveness properties of fair exchange, as well as ensures that the probability of violating fairness can be made arbitrarily low. We supply the probability distribution that optimizes the average complexity of the algorithm, in terms of its number of communication rounds, and we show that

the probability of violating fairness is inversely proportional to the average algorithm complexity.

Our algorithm is made of three phases, and we give the intuition underlying each phase below.

1. In the first phase, which we call the [illegible] phase, the security modules exchange the items that are supposed to be traded by their untrusted hosts. These items are not delivered by the security modules to their untrusted hosts: this is only performed if the third phase (below) terminates [illegible]. Any security module can decide here to abort the exchange if some item is missing or does not match its expected description. The security module hosted by the party that initiates the exchange also selects here a [illegible] k that it disseminates to all other security modules. The role of this random number is crucial in the second phase of the algorithm.
2. In the second phase, which we call the [illegible] phase, all security modules exchange messages during k rounds; each round following the same communication pattern as in the third phase (below). The fact that the random number k, determined in the first phase, is not accessible to the untrusted parties is fundamental here. Roughly speaking, the goal of the [illegible] phase is to make the probability, for [illegible] number of dishonest parties to successfully guess when the actual agreement phase is taking place (third phase below), arbitrarily low. If any dishonest party drops a message towards a honest party in this [illegible] phase, the security module hosted by the latter simply aborts the exchange and forces other modules to abort the exchange as well, thus penalizing any dishonest host that might try to bias the exchange in its favor.
3. In the third phase, which we call the [illegible] phase, the security modules solve a problem we call [illegible]. In this problem, the processes (in our case the security modules) start from an initial binary value (a proposal) and need to decide on a final binary value: either to abort the exchange or commit it (and deliver the items to their untrusted hosts). Unlike in consensus [17], but like in NBAC (non-blocking atomic commit) [30,9], the problem is biased towards 0: no process can decide 1 if some process proposes 0 (to avoid trivial solutions, the processes are supposed to decide 1 if no process fails or proposes 0). The agreement aspect of this problem is however different from consensus and NBAC; we simply require here that, if some process decides 1, then no correct process decides 0. We consider an early stopping algorithm that solves this problem in a model with general omissions, along the lines of [28].

Underlying our main contribution, i.e., a new gracefully degrading fair exchange algorithm, we contribute in bridging the gap between security problems (fair exchange) and traditional distributed computing problems (consensus-like problems). We show indeed that deterministic fair exchange in a model with security modules is equivalent to biased consensus. By proving that biased consensus is impossible in a synchronous model [24] with general omission failures [29] if half of the processes can be faulty, we establish a lower bound for fair exchange in a model with tamper proof modules.

1.2 Roadmap

Section 2 defines our system model. Section 3 recalls the fair exchange problem, introduces biased consensus, and shows their equivalence in a model with security modules. We also state the impossibility of deterministic fair exchange without a honest majority, which motivates our notion of gracefully degrading fair exchange. Section 4 describes our gracefully degrading fair exchange algorithm and states its correctness. Section 5 concludes the paper by discussing related work.

2 Model

The system we consider is composed of a set of processes, some modeling untrusted hosts and the other modeling security modules. These processes communicate by exchanging messages.

2.1 Untrusted Hosts and Security Modules

More precisely, the set of processes we consider is divided into two disjoint classes: [illegible] (or simply [illegible]) and [illegible]. Two processes connected by a physical channel are said to be adjacent. We assume that there exists a fully connected communication topology between the hosts, i.e., any two hosts are adjacent. Furthermore, we assume that every host process P_A is adjacent to exactly one security module process G_A (i.e., there is a bijective mapping between security modules and hosts): we say that P_A is [illegible] G_A. No two security modules are adjacent. In other words, for any two security modules G_A and G_B to communicate, they need to do so through their hosts P_A and P_B. This indirection provides the abstraction of an overlay network at the level of security modules. We call the part of the system consisting of security modules, and the virtual communication links between them, the [illegible]. We also assume that every host (resp. security module) has the knowledge about the entire set of other hosts (resp. security modules) that participate in the protocol. We call the part of the system consisting of hosts and the communication links between them the [illegible]. The notion of association can be extended to systems, meaning that, for any given untrusted system, the [illegible] is the system consisting of all security modules associated to any host in that untrusted system.

2.2 Security Modules and Virtual Channels

Security modules are interconnected by a virtual communication network with bidirectional channels over the physical communication network among the hosts. For simplicity, we denote the participants processes (the security modules) by $G_1, \ldots, G_n$. We assume that between any two security modules G_i and G_j, the following properties are guaranteed: (1) Message contents remain secret from unauthorized entities; (2) If a message is delivered at G_j, then it was previously sent by G_i; (3) Replayed messages are detected; (4) Message contents are not tampered with during transmission, i.e., any change during transmission will be detected and the message will be discarded; (5) If a message is sent by G_i to G_j and G_j is ready to receive the message, then the message will be delivered at G_j within some known bound Δ on the waiting time.

2.3 Trust and Adversary Model

Security modules can be trusted by other security modules or hosts, and hosts cannot be trusted by anybody. Hosts may be malicious, i.e., they may actively try to fool a protocol by not sending any message, sending wrong messages, or even sending the right messages at the wrong time. We assume however that hosts are computationally bounded, i.e., brute force attacks on secure channels are not possible. A malicious host may inhibit [illegible] communication between its associated security module and the outside world, yielding a channel in which messages can be lost.

A host [illegible] if it does not correctly follow the prescribed algorithm and we say that the host is [illegible]. Otherwise it is said to be [illegible]. Misbehavior is unrestricted (but computationally bounded as we pointed out). Security modules always follow their protocol, but since their associated hosts can inhibit all communication, this results in a system model of security modules with unreliable channels (the model of [illegible] [29], i.e., where messages may not be sent or received). In such systems, misbehavior (i.e., failing to send or receive a message) is sometimes termed [illegible]. We call security modules associated with honest hosts [illegible], whereas those associated with dishonest hosts [illegible]. In a set of n hosts, we use t to denote a bound on the number of hosts which are allowed to misbehave and f the number of hosts which actually do misbehave ($f \leq t$). Sometimes we restrict our attention to the case where $t < n/2$, i.e., where a majority of hosts is assumed to be honest. We call this the [illegible]. Our model of the adversary is based on the strongest possible attack, the case in which all of the f dishonest hosts collude. We assume that adversary knows all the algorithms and probability distributions used.

3 Variations on Fair Exchange and Impossibility Results

In this section we recall the definition of fair exchange (FE), and we show that this problem, at the level of untrusted hosts, is in a precise sense equivalent to a problem that we call [illegible] (BC), at the level of the underlying security modules. Then, we state that biased consensus is impossible if half of the processes can be faulty and derive the impossibility of fair exchange if half (or more) of the hosts are dishonest. This motivates our definition of a weaker variant of fair exchange, the gracefully degrading FE.

3.1 Fair Exchange

Definition 1 (Fair Exchange). [illegible] fair exchange *(* [illegible] *)* [illegible]

– *(*[illegible]*)* [illegible]
– *(*[illegible]*)* [illegible]
– *(*[illegible]*)* [illegible] *(*[illegible]*)* [illegible] *(*[illegible]*)* [illegible]

In case a host terminates without receiving the expected item, that host receives an abort indication (denoted ⊥). The Timeliness property ensures that every honest host can be sure that at some point in time the algorithm will terminate. The Effectiveness property states what should happen if all goes well. Finally, the Fairness property postulates restrictions on the information flow for the case where something goes wrong in the protocol.[1] Note that the first precondition of the Fairness property ("if the desired item of any host does not match the description ... ") is very important. Without this condition, a "successful" outcome of the exchange would be possible even if an item does not match the expected description, which should clearly be considered unfair.

3.2 Biased Consensus

Consider the following variant of consensus, we call [illegible] in a model where processes can fail by general omissions [29].

Definition 2 (Biased Consensus). [illegible] biased consensus *(* [illegible] *)* [illegible]

– *(*[illegible]*)* [illegible]
– *(*[illegible]*)* [illegible]
– *(*[illegible]*)* [illegible]
– *(*[illegible]*)* [illegible]

Processes invoke [illegible] using primitive $BCpropose(vote)$, $vote$ being a binary value, 0 or 1. Possible decisions are also 0 ([illegible]) and 1 ([illegible]). [illegible] and [illegible] are the same as in NBAC, whereas the [illegible] is weaker than the [illegible] property of NBAC [30].

We show below that FE and BC are equivalent in our model.

[1] We use here the concept of information flow to define fairness in a way that cleanly separates the distinct classes of *safety*, *liveness*, and *security* properties in the specification of the problem [26].

```
FairExchange(myitem, description, source, destination) returns item {
    ⟨send myitem to destination over secure channel⟩
    timed wait for ⟨expected item i from source over secure channel⟩
    ⟨check description on i⟩
    if ⟨check succeeds and no timeout⟩
    then vote := 1 else vote := 0 endif
    result := BCpropose(vote)
    if result = 1 then return i else return ⟨abort⟩ endif
}
```

Fig. 2. Using biased consensus to implement fair exchange: code of every host

Theorem 1. [illegible]

[illegible] (1) Assume that we have a solution to BC in the security subsystem consisting of security modules $G_1, \ldots, G_n$. Now consider the algorithm depicted in Fig. 2. This is a wrapper around the BC solution that solves FE at the level of the hosts. In other words, it is a reduction of FE into BC in our model. In the algorithm, a host hands to the associated security module its item and the executable description of the desired item, as well as identifiers of the hosts with which items should be exchanged. The security module exchanges the item with its partners, then checks the received item (*initialization* phase). Finally all security modules agree on the outcome using BC (*agreement* phase). The proposal value for BC is 1 if the check was successful and no abort was requested by the host in the meantime. If BC terminates with the process deciding 1, then the security module releases the item to the host. We now discuss each of the properties of fair exchange. The [illegible] property of FE is guaranteed by the [illegible] property of BC and the synchronous model assumption. Consider [illegible] and assume that all participating hosts are honest and all items match their descriptions. All votes for BC will be 1. Now the [illegible] property of BC guarantees that all processes (recall that every process is correct) will decide 1 and subsequently return the item to their hosts. Consider now [illegible] and observe that, in our adversary model, no dishonest host can derive any useful information from merely observing messages exchanged over the secure channels. The only way to receive information is through the interface of the [illegible] procedure. If one item does not match the description at some process, then this process will engage in BC with a *vote* = 0 (note that this will happen even if the associated host is dishonest). [illegible] of BC implies that the exchange results in no process deciding 1, so none of the hosts receives anything from the exchange. Additionally, if some honest host receives nothing through the exchange, then the [illegible] property of BC implies that no host can receive anything.

(2) Conversely, BC can be implemented using FE by invoking at every process G_i.[2]:

[illegible]([illegible]$_i$) = [illegible]([illegible]$_i, 1, G_{i-1}, G_{i+1}$).

Here we assume that if FE returns $\langle abort \rangle$ instead of the item, BC returns 0. So the votes, being exchange items in this case, are exchanged in a circular fashion among security modules. It is not difficult to see that FE properties guarantee the properties of BC. This is immediate for [illegible] and [illegible]. Consider now [illegible] and assume that G_j proposes 0 to BC. The item description checking at G_{j+1} will fail and the first part of FE [illegible] ("If the desired item of any host does not match its description … ") guarantees that every process that decides in BC decides 0. The second part of [illegible] guarantees [illegible].[3] □

Theorem 2. [illegible] $\lceil \frac{n}{2} \rceil$ [illegible]

We omit the proof of the Theorem 2 due to the lack of space. The proof can be found in the full version of the paper [2].

A direct corollary of the Theorems 1 and 2 leads to derive the following result:

Theorem 3. [illegible]

3.3 Early Stopping Biased Consensus Algorithm

The BC algorithm we give here (in Fig. 3) is an adaptation of the early-stopping synchronous consensus algorithm of [28]. This algorithm solves BC if there is a majority of correct processes ($t < n/2$). It is early stopping in the sense that every process terminates in at most $min(f + 2, t + 1)$ rounds. There are mainly two differences with the consensus algorithm of [28]: (1) the processes agree on a [illegible] of initially proposed values rather then on a [illegible] of those (in the case of consensus); (2) we also introduce *dummy* messages to have a [illegible] [24], i.e., to have a uniform communication pattern in every round. In other words, where in the original algorithm of [28] process G_i did not send a message to process G_j in round r, in our algorithm process G_i sends a [illegible] message m to process G_j, but process G_j disregards m. Our solution assumes that all BC protocol messages have the same size.[4] The motivation for having a full information protocol and uniform message size, will be explained in Section 4.

[2] To be precise, G_1 invokes $FairExchange(vote_1, 1, G_n, G_2)$ and G_n invokes $FairExchange(vote_n, 1, G_{n-1}, G_1)$.

[3] Note that, in contrast to BC, FE satisfies an information-flow (i.e., security) property [26]. This is why it was necessary to argue about the special properties of security modules when reducing FE to BC and not vice versa.

[4] This could be implemented by meeting the maximal size of BC message by padding all message of BC algorithm to reach this size.

```
BCpropose($vote_i$) returns $decision$ {
1:  $\boldsymbol{Vote_i} := \perp^n$; $Vote_i[i] := vote_i$; $\boldsymbol{new_i} := \perp^n$; $new_i[i] := vote_i$;
2:  $locked_i := \emptyset$; $suspected_i := \emptyset$; % r=0 %
3:  for $r := 1, 2, ..., t+1$ do % $r$: round number %
4:  begin_ round
5:      foreach $p_j$ do
6:          if $p_j \in suspected_i$ then $dummy := 1$ else $dummy := 0$ endif
7:          send $(new_i, locked_i, dummy)$ to $G_j$
8:      enddo
9:      $\boldsymbol{new_i} := \perp^n$
10:     foreach $G_j \notin suspected_i$ do
11:         if $(\boldsymbol{new_j}, locked_j, dummy = 0)$ has been received from $G_j$ then
12:             foreach $m \in [1 \ldots n]$ do
13:                 if $(new_j[m] \neq \perp)$ and $(Vote_i[m] = \perp)$ then
14:                     $Vote_i[m] := new_j[m]$; $new_i[m] := new_j[m]$
15:                     $locked_i := locked_i \cup locked_j$
16:                 endif
17:             enddo
18:         else
19:             if $(G_j \notin locked_i)$ then $suspected_i := suspected_i \cup \{G_j\}$ endif
20:         endif
21:     enddo
22:     if $(|suspected_i| > t)$ then return (0) endif
23:     if $(G_i \notin locked_i)$ then
24:         if $(r > |suspected_i|)$ or $(locked_i \neq \emptyset)$ then $locked_i := locked_i \cup \{G_i\}$ endif
25:     else
26:         if $(|locked_i| > t)$ then $decide(\boldsymbol{Vote_i})$ endif
27:     endif
28: end_ round
29: $decide(\boldsymbol{Vote_i})$
}

Procedure $decide(\boldsymbol{Vote})$ returns $decision$ {

30: if $(\exists m, 1 \leq m \leq n,$ s.t.$(Vote[m] = \perp)$ or $(Vote[m] = 0))$ then
31:     return (0)
32: else
33:     return (1)
34: end
}
```

Fig. 3. Pseudocode of a synchronous, early stopping Biased Consensus algorithm: code of process G_i

The changes we make to the algorithm of [28] do not affect the correctness of the algorithm. The difference is that we introduce the procedure *decide*(***Vote***) (lines 30-34, Fig. 3), where ***Vote*** is a vector of initially proposed values processes agree on. Basically, every process G_i stores information about the value initially proposed by some process G_j at $Vote_i[j]$. If the process G_i does not know the value that process G_j proposed, then $Vote_i[j] = \perp$. Roughly, a correct process does not learn the value proposed by process G_j, if G_j is faulty. [5]

We now give and prove the properties of our BC algorithm. We do not prove here that all processes that invoke *decide*() function (lines 30-34) agree on the vector $\boldsymbol{Vote}$ (this includes all correct processes, if $t < n/2$). Reader interested in this proof should refer to [28]. As discussed above, our algorithm inherently satisfies this property, given $t < n/2$. To summarize, our algorithm satisfies [illegible], [illegible] and [illegible] properties of BC. Furthermore, it satisfies the [illegible] property:

– (Early Stopping) Every correct process decides in at most $min(f+2, t+1)$ rounds.

[illegible] property is inherited from the original algorithm. Consider [illegible]. Assume that all processes are correct and all propose 1. In this case, all processes agree on vector $\boldsymbol{Vote} = 1^n$. Therefore *decide*() returns 1 at every process. Consider now [illegible] and note that, if any process decides 1 it must have invoked *decide*() and $\boldsymbol{Vote} = 1^n$. This implies that every correct process invokes *decide*(), evaluates the same vector $\boldsymbol{Vote}$ that processes agreed on and, therefore, returns 1. Consider now [illegible]. If some process G_j proposed $vote_j = 0$ every process G_i that invokes *decide*() (if any) has $Vote_i[j] = 0$ or $Vote_i[j] = \perp$, as processes agree on the vector $\boldsymbol{Vote}$ and the coordinate j of $\boldsymbol{Vote}$ is either $\perp$ or $vote_j$. Therefore no process can decide 1. Note that [illegible] holds for any t.

3.4 Gracefully Degrading Fair Exchange

The impossibility of solving FE (deterministically) if half of the processes can be dishonest, motivates the introduction of the following variant of the problem.

Definition 3 (Gracefully Degrading Fair Exchange). [illegible] gracefully degrading fair exchange ([illegible]) [illegible]

– [illegible]

– [illegible]

– [illegible] ([illegible]) [illegible] p $(0 < p < 1)$ [illegible] $(1-p)$ [illegible]

[5] With different implementations of the *decide()* procedure, the algorithm solves different problems. For example, if *decide*($\boldsymbol{Vote}$) would return a minimum of all (non-$\perp$) coordinates, the algorithm would solve consensus, which is precisely the case in [28].

4 A Gracefully Degrading Fair Exchange Algorithm

4.1 Description

Our GDFE algorithm is described in Figure 4. We assume that all processes involved in the algorithm know each other. The process with the lowest number is the initiator. We also assume a synchronous communication model [24] in the security subsystem. Basically, our algorithm can be viewed as an extension of the algorithm of Figure 2, i.e., our reduction of deterministic fair exchange to biased consensus. However, whereas the algorithm of Figure 2 is made of an [illegible] phase followed by an [illegible] (BC) phase, the algorithm of Figure 4 introduces a [illegible] phase between these two phases. This is the key to graceful degradation, i.e., to minimizing the probability of unfairness in the case when $t \geq n/2$. Basically, we do not run the BC algorithm immediately after the exchange of items (i.e., unlike in Fig. 2), but at some randomly picked round. In the meantime the processes exchange [illegible] messages and, if necessary, react to the behavior of hosts. If any process detects a host misbehavior, i.e., a message omission, it aborts the algorithm immediately (line 15) and does not participate in BC.[6] It is important to notice that the underlying BC algorithm guarantees that no process decides 1 if some process does not participate in the algorithm (this missing process might have proposed 0). This is the way of penalizing any host that misbehaves in the first two phases of the algorithm.

The [illegible] property of the underlying BC algorithm is essential for minimizing the probability for the adversary to violate fairness (as we discuss in the next subsection): in short, the early stoping BC algorithm we consider has two vulnerable rounds: if the adversary misses them, BC and the corresponding exchange terminate successfully. In addition, the introduction of [illegible] messages within the BC algorithm is necessary to solve the security requirement of our gracefully degrading fair exchange algorithm.

In our BC algorithm of Fig. 3, every process in every round sends exactly one message to every other process, but some (*dummy*) messages are tagged to be disregarded by the receiving process. This is necessary in order to make sure that the adversary has no means to distinguish the [illegible] phase from the [illegible] phase (i.e., the BC algorithm), we make use of the same communication pattern in both phases, i.e., the same distribution and sizes of the exchanged messages: Every process sends a fixed-size message to every other process in every round, both in the [illegible] phase and in the BC algorithm. Messages in [illegible] phase are, therefore, padded before sending, to the size of BC message. Hence, the adversary is not able to determine when BC starts, neither by observing when security modules send and receive messages, nor by observing the size of these messages.

[6] [25] uses a similar idea of choosing a random number of rounds and hiding it from the adversary to solve probabilistic non-repudiation (which is a special form of probabilistic fair exchange).

```
GDFairExchange(myitem, description, source, destination) returns item {
01: if ⟨$G_i$ is initiator⟩ then % initialization phase - round 0
02:    ⟨pick a random number $k$ according to a given distribution⟩
03:    foreach $G_j \neq$ destination do send $(\perp, k)$ to $G_j$ enddo
04:    send $(myitem, k)$ to destination
05: else
06:    send $(myitem, \perp)$ to destination
07: endif
08: if ((item, $*$) has been received from source) and
    (item matches description) and (($*, k$) has been received from initiator) then
09:    $vote_i := 1$; $k_i := k$; $item_i := item$
10: else
11:    return ($\perp$)
12: endif
13: for $round := 1, 2, \ldots, k_i$ do % fake phase - $k$ rounds
14:    send ⟨padded $vote_i$⟩ to all
15:    if not(($vote$) has been received from all processes) then return ($\perp$) endif
16: enddo
17: $vote_i :=$ BCpropose($vote_i$) % agreement phase - Biased Consensus
18: if ($vote_i = 1$) then return ($item_i$) else return ($\perp$) endif
}
```

Fig. 4. Pseudocode of the Gracefully Degrading Fair Exchange algorithm: code of process G_i

4.2 Correctness of GDFE Algorithm

Theorem 4. [illegible]

[illegible] The [illegible] property is guaranteed by the fact that we consider a synchronous system and the [illegible] property of BC. Consider [illegible] and assume that all participating hosts are honest and all items match their descriptions. All security modules will enter and exit the [illegible] phase having [illegible] $= 1$, so all security modules will [illegible] 1. By the [illegible] property of BC every module returns 1 and subsequently returns the expected item to its host. Now we consider [illegible]. It is important here to recall that the security modules are tamper-proof and no information leaks from them apart from what is explicitly released through their interface. We first prove a preliminary lemma. For convenience, if the security module returns $\perp$ to its host, we say that security module aborts the GDFE algorithm.

Lemma 1. [illegible] G_j [illegible] i $(0 \leq i < k)$, [illegible] $i + 1$ [illegible]

[illegible] Because G_j has aborted the GDFE algorithm at the end of round i, no security module will receive G_j's [illegible] in round $i+1$. From line 15, it can be seen that every security module will abort the algorithm at latest at the end of round $i + 1$ (some modules might have aborted the algorithm in round i, like G_j). □

Consider the case in which the first misbehavior of some of the dishonest hosts occurs in the round i where $0 \leq i < k$ (misbehavior in round 0 includes the initiator's misbehavior or some dishonest host sending the wrong item). According to Lemma 1, by the end of the round $i + 1 \leq k$, all security modules will abort the algorithm, so [illegible] is preserved.

Note that Lemma 1 does not hold for the k-th round. Some dishonest hosts can cut the channels for the first time in that round in such way that some security modules receive all messages and some do not. Hence some modules will [illegible] 1 and others will abort the algorithm at the end of round k and will not participate in BC. Because the modules that invoked consensus cannot distinguish this run from the run in which some faulty module proposed 0 and failed immediately in such way that it did not send or receive any message, all security modules that had invoked BC will return 0. At the end, none of the hosts gets the item.

The last possibility is that the first misbehavior occurs during the execution of the BC. This means that every security module has proposed 1 to BC. If there is a majority of honest hosts, the [illegible] property of BC guarantees [illegible]. Indeed, [illegible] can be violated only if some security module returns the expected item to its host, while some correct security module returns $\perp$ to its honest host. From line 18, it is obvious that this would be possible only if some security module returns 1 from BC, while some correct security module returns 0 which contradicts the [illegible] property. If the adversary controls half or more of the hosts [illegible] could be violated if, and only if, the adversary cuts one of the first two rounds of BC. However, this could occur only if the adversary successfully guesses in which round BC starts. Indeed, because our BC algorithm is [illegible], in order to succeed, the adversary must cut one of the first two rounds of BC and this has to be its first misbehavior in a particular algorithm run. In the following, we prove that if this case occurs, i.e., if there is no honest majority, probability of unfairness can be made arbitrarily low by choosing an appropriate distribution of the random number of [illegible] rounds.

The number k of rounds in the second phase of the algorithm is chosen randomly by the initiator of the exchange according to a given distribution $(\beta_0, \beta_1, \dots)$ i.e., $\Pr(k = i) = \beta_i$. We assume this distribution to be public. The adversary performs the attack in a given round by dropping a certain subset of messages sent to, or received by, the hosts it controls, i.e., by cutting the channels. When the adversary cuts channels at more than $n/2$ hosts in the same round, we say that he [illegible]. Since the adversary does not know in which round BC starts, the best attack consists in choosing a value i according to the distribution $(\beta_0, \beta_1, \dots)$, starting from which adversary cuts all the rounds until the end of the exchange. Cutting messages at less that $n/2$ hosts, or cutting non-consecutive rounds, cannot improve the probability of success of the adversary.

We define the [illegible] $\Gamma_{(\beta_0, \beta_1, \dots)}$ as the maximum probability that an adversary succeeds, given the distribution $(\beta_0, \beta_1, \dots)$, and the [illegible] in terms of number of [illegible] rounds as $\Lambda_{(\beta_0, \beta_1, \dots)} = \sum_{i \geq 1} i\beta_i$.

Lemma 2. [illegible] $(\beta_0, \beta_1, \dots)$ [illegible] k [illegible] ([illegible]) [illegible]

$$\varGamma_{(\beta_0,\beta_1,\ldots)} = \max_{i\geq 0}(\beta_i + \beta_{i+1}).$$

[illegible] Let γ_i be the probability that the attack succeeds if it starts at round i ($i > 0$). We already know that $\gamma_{i\leq k} = 0$ and that $\gamma_{i>k+2} = 0$. We have therefore:

$$\gamma_1 = \beta_0,\ \gamma_2 = \beta_0 + \beta_1,\ \gamma_3 = \beta_1 + \beta_2, \ldots,\ \gamma_i = \beta_{i-2} + \beta_{i-1}, \ldots$$

According to the probability distribution $(\beta_0, \beta_1, \ldots)$, the maximum probability of unfairness $\varGamma_{(\beta_0,\beta_1,\ldots)}$ is therefore $\varGamma_{(\beta_0,\beta_1,\ldots)} = \max_{i>0}(\gamma_i) = \max_{i\geq 2}(\beta_0, \beta_{i-2} + \beta_{i-1}) = \max_{i\geq 0}(\beta_i + \beta_{i+1})$. □

We define the probability distribution that we call [illegible], as well as the [illegible] probability distribution for the algorithm of Figure 4.

Definition 4. [illegible] $(\beta_0, \beta_1, \ldots)$ [illegible] t [illegible] $[0, \kappa]$ [illegible] $\forall i \geq 0,\ \beta_i + \beta_{i+1} = \frac{1}{\lceil \frac{\kappa+1}{2} \rceil}$ [illegible] $\beta_1 = t$ [illegible] κ [illegible] $\beta_1 = 0$ [illegible] κ [illegible]

Definition 5. [illegible] $(\beta_0, \beta_1, \ldots)$ [illegible] ([illegible]) [illegible] $(\beta'_0, \beta'_1, \ldots)$ [illegible] $\exists \varGamma > 0$ [illegible] $\forall i \geq 0,\ \beta_i + \beta_{i+1} \leq \varGamma,\ \beta'_i + \beta'_{i+1} \leq \varGamma$ [illegible] $\varLambda_{(\beta'_0,\beta'_1,\ldots)} < \varLambda_{(\beta_0,\beta_1,\ldots)}$

The following lemma states our optimality result in terms of probability of unfairness.

Lemma 3. [illegible] ([illegible]) [illegible] $[0, \kappa]$ [illegible] κ [illegible] $\varGamma_{bi\text{-}uniform} = \frac{2}{\kappa+2}$ [illegible] $\varLambda_{bi\text{-}uniform} = \frac{\kappa}{2}$ [7]

Due to the lack of space, we omit here the formal proof of Lemma 3. The proof can be found in the full version of the paper [2]. □

5 Concluding Remarks

It was shown in [16] that deterministic two-party fair exchange is impossible without a trusted third party. Published results on multi-party protocols focus

[7] It can be shown that our GDFE algorithm satisfies $\varGamma_{\text{bi-uniform}}/\varXi_{\text{bi-uniform}} \approx 2/\varLambda_{\text{bi-uniform}}$, where $\varXi$ is the probability that all processes successfully terminate the algorithm; rephrasing the result of [33] in our context would mean that the optimal for a randomized biased consensus is $\varGamma/\varXi \geq 1/\varLambda$ (instead of $2/\varLambda$). We claim that the factor 2 in our case is due to the fact that we ensure biased agreement with a correct majority.

on reducing the necessary trust in the third party [20, 7], or on contract signing [21, 8], a special form of fair exchange. In [22] it was shown that, in a synchronous system with computational security, a majority of honest processes can simulate a centralized trusted third party (and hence solve fair exchange) using cryptography. In a somewhat stronger model, [18] gives the solution for secure multi-party computation in a synchronous system with unconditional security, that also assumes a majority of honest processes. The use of security modules in fair exchange was already explored in the two-party context: in particular, [34] employs smart cards as security modules to solve two-party fair exchange in an optimistic way, whereas [3] describes a probabilistic solution to two-party fair exchange.[8] Idea of using a distributed trusted third party in solving two-party fair exchange was exploited in [4].

Recent works [23, 19] have solved various forms of secure multi-party computation (SMPC) for any number of dishonest parties ($t < n$). As fair exchange is usually considered as a special case of a SMPC, it can be tempting to conclude that these results also apply to fair exchange. However, in the relaxed definitions of SMPC considered in [23] and, implicitly in [19], [illegible] is not always required, as discussed in [23]. We consider in this paper contexts where [illegible] is mandatory. In this sense, results shown in this paper are rather complementary to those that establish the possibility of solving SMPC for any $t < n$.

Acknowledgments

We are very grateful to Bastian Pochon for his useful comments on the specification of biased consensus. Gildas Avoine is supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

References

1. N. Asokan, M. Schunter, and M. Waidner. Optimistic protocols for fair exchange. In T. Matsumoto, editor, *4th ACM Conference on Computer and Communications Security*, pages 8–17, Zurich, Switzerland, Apr. 1997. ACM Press.
2. G. Avoine, F. Gärtner, R. Guerraoui, and M. Vukolić. Gracefully degrading fair exchange with security modules. Technical Report IC/2004/26, EPFL, Switzerland, 2004.

[8] Two-party fair exchange is clearly simpler than the multi-party fair exchange. Informally, a deterministic solution with honest majority in the two-party case, i.e., when both (all) participating hosts are honest, is possible in a single communication round. On the other hand, two rounds of communication are minimum for the deterministic solution to biased consensus (and, therefore, for fair exchange) with the honest majority.

3. G. Avoine and S. Vaudenay. Fair exchange with guardian angels. In *The 4th International Workshop on Information Security Applications – WISA 2003*, Lecture Notes in Computer Science, Jeju Island, Korea, August 2003. Springer-Verlag.
4. G. Avoine and S. Vaudenay. Optimistic fair exchange based on publicly verifiable secret sharing. In *Proceedings of the 9th Australasian Conference on Information Security and Privacy*, Lecture Notes in Computer Science, pages 74–85, Sydney, Australia, 2004. Springer-Verlag.
5. M. Backes, B. Pfitzmann, M. Steiner, and M. Waidner. Polynomial fairness and liveness. In *15th IEEE Computer Security Foundations Workshop – CSFW*, pages 160–174, Cape Breton, Nova Scotia, Canada, June 2002. IEEE, IEEE Computer Society.
6. F. Bao, R. Deng, and W. Mao. Efficient and practical fair exchange protocols with off-line TTP. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 77–85, Oakland, California, USA, May 1998. IEEE Computer Society Press.
7. F. Bao, R. Deng, K. Q. Nguyen, and V. Varadharajan. Multi-party fair exchange with an off-line trusted neutral party. In *Proceedings of the 10th International Workshop on Database & Expert Systems Applications*, pages 858–862, Florence, Italy, 1–3 Sept. 1999. IEEE Computer Society Press.
8. B. Baum-Waidner and M. Waidner. Round-optimal and abuse-free multi-party contract signing. In *27th International Colloquium on Automata, Languages and Programming (ICALP '2000)*, volume 1853 of *Lecture Notes in Computer Science*, pages 524–535, Geneva, Switzerland, July 2000. Springer-Verlag.
9. P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.
10. D. Boneh and M. Naor. Timed commitments. In M. Bellare, editor, *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 236–254, Santa Barbara, California, USA, August 2000. IACR, Springer-Verlag.
11. C. Boyd and E. Foo. Off-line fair payment protocols using convertible signatures. In K. Ohta and D. Pei, editors, *Advances in Cryptology – ASIACRYPT'98*, volume 1514 of *Lecture Notes in Computer Science*, pages 271–285, Beijing, China, October 1998. IACR, Springer-Verlag.
12. C. Boyd and P. Kearney. Exploring fair exchange protocols using specification animation. In J. Pieprzyk, E. Okamoto, and J. Seberry, editors, *Proceedings of ISW 2000*, volume 1975 of *Lecture Notes in Computer Science*, pages 209–223, Wollongong, Australia, December 2000. Springer-Verlag.
13. H. Bürk and A. Pfitzmann. Value exchange systems enabling security and unobservability. *Computers & Security*, 9(8):715–721, 1990.
14. L. Chen. Efficient fair exchange with verifiable confirmation of signatures. In K. Ohta and D. Pei, editors, *Advances in Cryptology – ASIACRYPT'98*, volume 1514 of *Lecture Notes in Computer Science*, pages 286–299, Beijing, China, October 1998. IACR, Springer-Verlag.
15. J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, Oct. 2001.
16. S. Even and Y. Yacobi. Relations amoung public key signature systems. Technical Report 175, Computer Science Department, Technicon, Haifa, Israel, 1980.
17. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

18. M. Fitzi, N. Gisin, U. M. Maurer, and O. von Rotz. Unconditional byzantine agreement and multi-party computation secure against dishonest minorities from scratch. In *Proceedings of the 21st annual symposium on the Theory and Applications of Cryptographic Techniques*, pages 482–501. Springer-Verlag, 2002.
19. M. Fitzi, D. Gottesman, M. Hirt, T. Holenstein, and A. Smith. Detectable byzantine agreement secure against faulty majorities. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 118–126. ACM Press, 2002.
20. M. K. Franklin and G. Tsudik. Secure group barter: Multi-party fair exchange with semi-trusted neutral parties. In *Financial Cryptography – FC '98*, volume 1465 of *Lecture Notes in Computer Science*, pages 90–102, Anguilla, British West Indies, Feb. 1998. Springer-Verlag.
21. J. A. Garay and P. MacKenzie. Abuse-free multi-party contract signing. In *Distributed Computing – DISC '99*, volume 1693 of *Lecture Notes in Computer Science*, pages 151–165, Bratislava, Slovak Rep., 27–29 Sept. 1999. Springer-Verlag.
22. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game — a completeness theorem for protocols with honest majority. In *Proceedings of the 19th ACM Symposium on the Theory of Computing (STOC)*, pages 218–229, 1987.
23. S. Goldwasser and Y. Lindell. Secure computation without agreement. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 17–32. Springer-Verlag, 2002.
24. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Mateo, CA, 1996.
25. O. Markowitch and Y. Roggeman. Probabilistic non-repudiation without trusted third party. In *Proceedings of the 2nd Workshop on Security in Communication Networks*, 1999.
26. J. McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 22(1):53–67, Jan. 1996. Special Section—Best Papers of the IEEE Symposium on Security and Privacy 1994.
27. H. Pagnia, H. Vogt, and F. C. Gärtner. Fair exchange. *The Computer Journal*, 46(1), 2003.
28. P. R. Parvédy and M. Raynal. Optimal early stopping uniform consensus in synchronous systems with process omission failures. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 302–310. ACM Press, 2004.
29. K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, Mar. 1986.
30. D. Skeen. Non-blocking commit protocols. In *Proc. ACM SIGMOD Conf.*, page 133, Ann Arbor, MI, Apr.-May 1981.
31. T. Tedrick. Fair exchange of secrets. In G. R. Blakley and D. Chaum, editors, *Advances in Cryptology – CRYPTO'84*, volume 196 of *Lecture Notes in Computer Science*, pages 434–438, Santa Barbara, California, USA, August 1985. IACR, Springer-Verlag.
32. Trusted Computing Group. Trusted computing group homepage. Internet: `https://www.trustedcomputinggroup.org/`, 2003.
33. G. Varghese and N. A. Lynch. A tradeoff between safety and liveness for randomized coordinated attack. *Information and Computation*, 128(1):57–71, 1996.
34. H. Vogt, F. C. Gärtner, and H. Pagnia. Supporting fair exchange in mobile environments. *ACM/Kluwer Journal on Mobile Networks and Applications (MONET)*, 8(2), Apr. 2003.

Adding Fault-Tolerance Using Pre-synthesized Components[1]

Sandeep S. Kulkarni and Ali Ebnenasir

Department of Computer Science and Engineering,
Michigan State University,
48824 East Lansing, Michigan, USA
{sandeep, ebnenasi}@cse.msu.edu
http://www.cse.msu.edu/~{sandeep,ebnenasi}

Abstract. We present a hybrid synthesis method for automatic addition of fault-tolerance to *distributed* programs. In particular, we automatically specify and add pre-synthesized fault-tolerance components to programs in the cases where existing heuristics fail to add fault-tolerance. Such addition of pre-synthesized components has the advantage of *reusing* pre-synthesized fault-tolerance components in the synthesis of different programs, and as a result, reusing the effort put in the synthesis of one program for the synthesis of another program. Our synthesis method is sound in that the synthesized fault-tolerant program satisfies its specification in the absence of faults, and provides desired level of fault-tolerance in the presence of faults. We illustrate our synthesis method by adding pre-synthesized components with linear topology to a token ring program that tolerates the corruption of all processes. Also, we have reused the same component in the synthesis of a fault-tolerant alternating bit protocol. Elsewhere, we have applied this method for adding presynthesized components with hierarchical topology.

Keywords: Automatic addition of fault-tolerance, Formal methods, Detectors, Correctors, Distributed programs.

1 Introduction

Automatic synthesis of fault-tolerant distributed programs from their fault-intolerant versions is desirable in variety of disciplines (e.g., safety-critical systems, embedded systems, network protocols) since such automated synthesis (i) generates a program that is correct by construction, and (ii) has the potential to preserve the properties of the fault-intolerant program. However, the exponential complexity of synthesis is one of the important obstacles in such automated synthesis. Thus, it is desirable to reuse the effort put in the synthesis of one program for the synthesis of another program. In this paper, we concentrate on the

[1] This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

identification and the addition of pre-synthesized fault-tolerance components to fault-intolerant programs so that we can reuse those components in the synthesis of different programs.

In the previous work on automatic transformation of fault-intolerant programs to fault-tolerant programs, Kulkarni and Arora [1] present polynomial time algorithms (in the state space of the fault-intolerant program) for the synthesis of fault-tolerant programs in the high atomicity model – where each process of the program can read/write all program variables in an atomic step. However, for the synthesis of fault-tolerant distributed programs, they show that the complexity of synthesis is exponential. Techniques presented in [2, 3, 4] reduce the complexity of synthesis by using heuristics and by identifying classes of programs and specifications for which efficient synthesis is possible. However, these approaches cannot apply the lessons learnt in synthesizing one fault-tolerant program while synthesizing another fault-tolerant program. As we encounter new problems, it is desirable to reuse synthesis techniques that we have already used during the synthesis of other problems. Hence, if we recognize the [illegible] that we often apply in the synthesis of fault-tolerant distributed programs then we can organize those patterns in terms of fault-tolerance components and reuse them in the synthesis of new problems.

To investigate the use of pre-synthesized fault-tolerance components in the synthesis of fault-tolerant distributed programs, we use [illegible] and [illegible] identified in [5]. Specifically, in [5], it is shown that detectors and correctors suffice in the [illegible] design of a rich class of [illegible] fault-tolerant programs – where the fault-tolerant program satisfies its safety and liveness specification even in the presence of faults. To achieve our goal, we present a synthesis method that adds pre-synthesized detectors and correctors to a given fault-intolerant program in order to synthesize its fault-tolerant version. Using our synthesis method, we identify (i) the representation of the pre-synthesized detectors and correctors; (ii) when and where the synthesis algorithm should use a detector or a corrector, and (iii) how to ensure the correctness of the fault-tolerant program and pre-synthesized detectors and correctors in the presence of each other.

Contributions. The contributions of this paper are as follows: (i) we develop a synthesis method for reusing pre-synthesized fault-tolerance components in the synthesis of different programs; (ii) we reduce the chance of failure of the synthesis algorithm by using pre-synthesized fault-tolerance components in the cases where existing heuristics fail; (iii) we present a systematic approach for expanding the state space of the program being synthesized in the cases where synthesis fails in the original state space, and finally (iv) we present a systematic method for adding new variables to programs for the sake of adding fault-tolerance.

As an illustration of our synthesis method, we add pre-synthesized components with linear topology to a token ring program that is subject to process-restart faults. The masking fault-tolerant (token ring) program can recover even from the situation where every process is corrupted. We note that the previous approaches that added fault-tolerance to the token ring program presented in this paper fail to synthesize a fault-tolerant program when all processes are corrupted. We have also synthesized a fault-tolerant alternating bit protocol by [illegible] the same pre-synthesized fault-tolerance component used in the synthesis of the token ring program (cf. [6] for this synthesis). Elsewhere [7], we have used this method for synthesizing a fault-tolerant diffusing computation where the added component is hierarchical in nature. This example also demonstrates the addition of multiple components. Thus, the synthesis method presented in this paper can be used for adding fault-tolerance components (i) on different topologies, and (ii) for different types of faults.

[illegible] The notion of program in this paper refers to the abstract structure of a program. The abstract structure of a program is an abstraction of the parts of the program code that execute inter-process synchronization tasks.

The Organization of the Paper. In Section 2, we present preliminary concepts. In Section 3, we formally state the problem of adding fault-tolerance components to fault-intolerant programs. Then, in Section 4, we present a synthesis method that identifies when and how the synthesis algorithm decides to add a component. Subsequently, in Section 5, we describe how we formally represent a fault-tolerance component. In Section 6, we show how we automatically specify a required component and add it to a program. We discuss issues related to our synthesis method in Section 7. Finally, we make concluding remarks and discuss future work in Section 8.

2 Preliminaries

In this section, we give formal definitions of programs, problem specifications, faults, and fault-tolerance. The programs are specified in terms of their state space and their transitions. We have adapted the definition of (i) specifications from Alpern and Schneider [8], and (ii) faults and fault-tolerance from Arora and Gouda [9] and Kulkarni and Arora [10]. The issues of modeling distributed programs is adapted from [1,11].

Program. A program p is a finite set of variables and a finite set of processes. Each variable is associated with a finite domain of values. A state of p is obtained by assigning each variable a value from its respective domain. The state space of p, S_p, is the set of all possible states of p.

A process, say P_j, in p is associated with a set of program variables, say r_j, that P_j can read and a set of variables, say w_j, that P_j can write. Also, process P_j consists of a set of transitions of the form (s_0, s_1) where $s_0, s_1 \in S_p$. The set of the transitions of p is the union of the transitions of its processes.

A state predicate of p is any subset of S_p. A state predicate S is closed in the program p iff (if and only if) $\forall s_0, s_1 : (s_0, s_1) \in p : (s_0 \in S \Rightarrow s_1 \in S)$. A sequence of states, $\langle s_0, s_1, ... \rangle$, is a computation of p iff the following two conditions are satisfied: (1) $\forall j : j > 0 : (s_{j-1}, s_j) \in p$, and (2) if $\langle s_0, s_1, ... \rangle$ is finite and terminates in state s_l then there does not exist state s such that $(s_l, s) \in p$. A sequence of states, $\langle s_0, s_1, ..., s_n \rangle$, is a computation prefix of p iff $\forall j : 0 < j \leq n : (s_{j-1}, s_j) \in p$, i.e., a computation prefix need not be maximal. The projection of program p

on state predicate S, denoted as $p|S$, consists of transitions $\{(s_0, s_1) : (s_0, s_1) \in p \ \wedge \ s_0, s_1 \in S\}$.

Distribution Issues. We model distribution by identifying how read/write restrictions on a process affect its transitions. A process P_j cannot include transitions that write a variable x, where $x \notin w_j$. In other words, the write restrictions identify the set of transitions that a process P_j can execute. Given a single transition (s_0, s_1), it appears that all the variables must be read to execute that transition. For this reason, read restrictions require us to group transitions and ensure that the entire group is included or the entire group is excluded. As an example, consider a program consisting of two variables a and b, with domains $\{0, 1\}$. Suppose that we have a process that cannot read b. Now, observe that the transition from the state $\langle a = 0, b = 0\rangle$ to $\langle a = 1, b = 0\rangle$ can be included iff the transition from $\langle a = 0, b = 1\rangle$ to $\langle a = 1, b = 1\rangle$ is also included. If we were to include only one of these transitions, we would need to read both a and b. However, when these two transitions are grouped, the value of b is irrelevant, and we do not need read it.

Specification. A specification is a set of infinite sequences of states that is suffix closed and fusion closed. Suffix closure of the set means that if a state sequence σ is in that set then so are all the suffixes of σ. Fusion closure of the set means that if state sequences $\langle\alpha, s, \gamma\rangle$ and $\langle\beta, s, \delta\rangle$ are in that set then so are the state sequences $\langle\alpha, s, \delta\rangle$ and $\langle\beta, s, \gamma\rangle$, where α and β are finite prefixes of state sequences, γ and δ are suffixes of state sequences, and s is a program state.

Following Alpern and Schneider [8], we rewrite the specification as the intersection of a safety specification and a liveness specification. For a suffix-closed and fusion-closed specification, the safety specification can be specified [10] as a set of bad transitions that must not occur in program computations, that is, for program p, its safety specification is a subset of $S_p \times S_p$.

Given a program p, a state predicate S, and a specification *spec*, we say that p satisfies *spec* from S iff (1) S is closed in p, and (2) every computation of p that starts in a state in S is in *spec*. If p satisfies *spec* from S and $S \neq \{\}$, we say that S is an invariant of p for spec.

We do not explicitly specify the liveness specification in our algorithm; the liveness requirements for the synthesis is that the fault-tolerant program eventually recovers to states from where it satisfies its safety and liveness specification.

Faults. The faults that a program is subject to are systematically represented by transitions. A fault f for a program p with state space S_p, is a subset of the set $S_p \times S_p$. A sequence of states, $\sigma = \langle s_0, s_1, ...\rangle$, is a computation of p in the presence of f (denoted $p[]f$) iff the following three conditions are satisfied: (1) every transition $t \in \sigma$ is a fault or program transition; (2) if σ is finite and terminates in s_l then there exists no program transition originating at s_l, and (3) the number of fault occurrences in σ is finite.

We say that a state predicate T is an f-span (read as fault-span) of p from S iff the following two conditions are satisfied: (1) $S \Rightarrow T$ and (2) T is closed in $p[]f$. Observe that for all computations of p that start at states where S is true,

T is a boundary in the state space of p up to which (but not beyond which) the state of p may be perturbed by the occurrence of the transitions in f.

Fault-Tolerance. Given a program p, its invariant, S, its specification, *spec*, and a class of faults, f, we say p is masking f-tolerant for *spec* from S iff the following two conditions hold: (i) p satisfies *spec* from S; (ii) there exists a state predicate T such that T is an f-span of p from S, $p[]f$ satisfies *spec* from T, and every computation of $p[]f$ that starts from a state in T has a state in S.

[illegible] We use Dijkstra's guarded commands [12] to represent the set of program transitions. A guarded command (action) is of the form $grd \rightarrow st$, where grd is a state predicate and st is a statement that updates the program variables. The guarded command $grd \rightarrow st$ includes all program transitions $\{(s_0, s_1) : grd$ holds at s_0 and the [illegible] execution of st at s_0 takes the program to state $s_1\}$.

3 Problem Statement

In this section, we formally define the problem of adding fault-tolerance components to a fault-intolerant program. We identify the conditions of the addition problem by which we can verify the correctness of the synthesized fault-tolerant program after adding fault-tolerance components.

Given a fault-intolerant program p, its state space S_p, its invariant S, its specification *spec*, and a class of faults f, we add pre-synthesized fault-tolerance components to p in order to synthesize a fault-tolerant program p' with the new invariant S'. When we add a fault-tolerance component to p, we also add the variables associated with that component. As a result, we expand the state space of p. The new state space, say $S_{p'}$, is actually the state space of the synthesized fault-tolerant program p'.

After the addition, we require the fault-tolerant program p' to behave similar to p in the absence of faults f. In the presence of faults f, p' should satisfy [illegible] fault-tolerance. To ensure the correctness of the synthesized fault-tolerant program in the new state space, we need to identify the conditions that have to be met by the synthesized program, p'. Towards this end, we define a projection from $S_{p'}$ to S_p using onto function $H : S_{p'} \rightarrow S_p$. We apply H on states, state predicates, transitions, and groups of transitions in $S_{p'}$ to identify their corresponding entities in S_p.

Let the invariant of the synthesized program be $S' \subseteq S_{p'}$. If there exists a state $s'_0 \in S'$ where $H(s'_0) \notin S$ then in the absence of faults p' can start at s'_0 whose image, $H(s'_0)$, is outside S. As a result, in the absence of faults, p' will include computations in the new state space $S_{p'}$ that do not have corresponding computations in p. These new computations resemble new behaviors in the absence of faults, which is not desirable. Therefore, we require that $H(S') \subseteq S$. Also, if p' contains a transition (s'_0, s'_1) in $p'|S'$ that does not have a corresponding transition (s_0, s_1) in $p|H(S')$ (where $H(s'_0) = s_0$ and $H(s'_1) = s_1$) then p' can take this transition and create a new way for satisfying *spec* in the absence of faults. Therefore, we require that $H(p'|S') \subseteq p|H(S')$. Now, we present the problem of adding fault-tolerance components to p.

The Addition Problem.
Given p, S, *spec*, f, with state space S_p such that p satisfies *spec* from S,
  $S_{p'}$ is the new state space due to adding fault-tolerance components to p,
  $H : S_{p'} \rightarrow S_p$ is an onto function,
Identify p' and $S' \subseteq S_{p'}$ such that
  $H(S') \subseteq S$,
  $H(p'|S') \subseteq p|H(S')$, and
  p' is masking f-tolerant for *spec* from S'. $\square$

4 The Synthesis Method

In this section, we present a synthesis method to solve the addition problem. In Section 4.1, we present a high level description of our synthesis method and express our approach for combining heuristics from [2] (cf. Section 4.2 for an example heuristic) with pre-synthesized components. Then, in Section 4.2, we illustrate our synthesis method using a simple example, a token ring program with 4 processes. We use the token ring program as a running example in the rest of the paper, where we synthesize a token ring program that is masking fault-tolerant to process-restart faults.

4.1 Overview of Synthesis Method

Our synthesis method takes as its input a fault-intolerant program p with a set of processes $P_0 \cdots P_n$ $(n > 1)$, its specification *spec*, its invariant S, a set of read/write restrictions $r_0 \cdots r_n$ and $w_0 \cdots w_n$, and a class of faults f to which we intend to add fault-tolerance. The synthesis method outputs a fault-tolerant program p' and its invariant S'.

The heuristics in [2] (i) [illegible] to ensure that the masking fault-tolerant program never violates its safety specification, and (ii) [illegible] to ensure that the masking fault-tolerant program never deadlocks (respectively, livelocks). Moreover, while adding recovery transitions, it is necessary to ensure that all the groups of transitions included along that recovery transition are safe unless it can be guaranteed (with the help from heuristics) that those transitions cannot be executed. Thus, adding recovery transitions from deadlock states is one of the important issues in adding fault-tolerance. Hence, the method presented in this paper, focuses on adding pre-synthesized components for resolving such deadlock states, say s_d.

Now, in order to resolve s_d using our hybrid approach, we proceed as follows: First, for each process P_i in the given fault-intolerant program, we introduce a high atomicity pseudo process PS_i. Initially, PS_i has no action to execute, however, we allow PS_i to read all program variables and write only those variables that P_i can write. Using these special processes, we present the *ResolveDeadlock* routine (cf. Figure 1) that is the core of our synthesis method. The input of

ResolveDeadlock consists of the deadlock state that needs to be resolved, s_d, and the set of high atomicity pseudo processes PS_i $(0 \leq i \leq n)$.

First, in Step 1, we invoke a heuristic-based routine *Add_Recovery* to add recovery from s_d under the distribution restrictions (i.e., in the low atomicity model) – where program processes have read/write restrictions with respect to the program variables. *Add_Recovery* explores the ability of each process P_i to add recovery transition from s_d under the distribution restrictions. If *Add_Recovery* fails then we will choose to add a fault-tolerance component in Steps 2 and 3.

In Steps 2 and 3, we identify a fault-tolerance component and then add it to p in order to resolve s_d. To add a fault-tolerance component, the synthesis algorithm should (i) specify the required component; (ii) retrieve the specified component from a given library of components; (iii) ensure the interference freedom of the component and the program, and finally (iv) add the extracted component to the program. As a result, adding a pre-synthesized component is a costly operation. Hence, we prefer to add a component during the synthesis only when available heuristics for adding recovery fail in Step 1.

```
Resolve_Deadlock($s_d$: state, $PS_0, \cdots, PS_n$: high atomicity pseudo process)
{
  Step 1. If Add_Recovery ($s_d$) then return [illegible].
  Step 2. Else non-deterministically choose a $PS_{index}$, where $0 \leq index \leq n$ and $PS_{index}$
            adds a high atomicity recovery action $grd \rightarrow st$
  Step 3. If (there exists a $PS_{index}$) and (there exists a detector $d$ in the component
            library that suffices to refine $grd \rightarrow st$ without interfering with the program)
            then add $d$ to the program, and return [illegible].
            else return [illegible].
               // Subsequently, we remove some transitions to make $s_d$ unreachable.
}
```

Fig. 1. Overview of the synthesis method

To identify the required fault-tolerance components, we use pseudo process PS_i that can read all program variables and write w_i (i.e., the set of variables that P_i can write). In other words, we check the ability of each PS_i to add high atomicity recovery – where we have no read restrictions – from s_d. If no PS_i can add recovery from s_d then our algorithm fails to resolve s_d. If there exist one or more pseudo processes that add recovery from s_d then we non-deterministically choose a process PS_{index} with high atomicity action $ac : grd \rightarrow st$. Since we give PS_{index} the permission to read all program variables for adding recovery from s_d, the guard grd is a global state predicate that we need to refine. If there exists a detector that can refine grd without interfering with the program execution then we will add that detector to the program. (The discussion about how to specify the required detector d and how to add d to the fault-intolerant program is in Sections 5 and 6.)

In cases where *Resolve_Deadlock* returns [illegible], we remove some transitions to make s_d unreachable. If we fail to make s_d unreachable then we will declare failure in the synthesis of the masking fault-tolerant program p'. Observe that by using pre-synthesized components, we increase the chance of adding recovery from s_d, and as a result, we reduce the chance of reaching a point where we declare failure to synthesize a fault-tolerant program.

4.2 Token Ring Example

Using our synthesis method (cf. Figure 1), we synthesize a token ring program that is masking fault-tolerant for the case where all processes are corrupted.

The Token Ring Program. The fault-intolerant program consists of four processes P_0, P_1, P_2, and P_3 arranged in a ring. Each process P_i has a variable x_i $(0 \leq i \leq 3)$ with the domain $\{\perp, 0, 1\}$. Due to distribution restrictions, process P_i can read x_i and x_{i-1} and can only write x_i $(1 \leq i \leq 3)$. P_0 can read x_0 and x_3 and can only write x_0. We say, a process P_i $(1 \leq i \leq 3)$ has the token iff $x_i \neq x_{i-1}$ and fault transitions have not corrupted P_i and P_{i-1}. And, P_0 has the token iff $x_3 = x_0$ and fault transitions have not corrupted P_0 and P_3. A process P_i $(1 \leq i \leq 3)$ copies x_{i-1} to x_i if the value of x_i is different from x_{i-1}. Also, if $x_0 = x_3$ then process P_0 copies the value of $(x_3 \oplus 1)$ to x_0, where $\oplus$ is addition in modulo 2. This way, a process passes the token to the next process.

We denote a state s of the token ring program by a 4-tuple $\langle x_0, x_1, x_2, x_3 \rangle$. Each element of the 4-tuple $\langle x_0, x_1, x_2, x_3 \rangle$ represents the value of x_i in s $(0 \leq i \leq 3)$. Thus, if we start from initial state $\langle 0, 0, 0, 0 \rangle$ then process P_0 has the token and the token circulates along the ring. We represent the transitions of the fault-intolerant program TR by the following actions $(1 \leq i \leq 3)$.

$$\begin{array}{lll}
TR_0: & (x_0 = 1) \wedge (x_3 = 1) & \longrightarrow x_0 := 0; \\
TR'_0: & (x_0 = 0) \wedge (x_3 = 0) & \longrightarrow x_0 := 1; \\
TR_i: & (x_i = 0) \wedge (x_{i-1} = 1) & \longrightarrow x_i := 1; \\
TR'_i: & (x_i = 1) \wedge (x_{i-1} = 0) & \longrightarrow x_i := 0;
\end{array}$$

[illegible] Faults can restart a process P_i. Thus, the value of x_i becomes unknown. We use $\perp$ to model the unknown value of x_i.

[illegible] The problem specification requires that the corrupted value of one process does not affect a non-corrupted process, and there is only one process that has the token.

[illegible] The invariant of the above program includes states $\langle 0,0,0,0 \rangle$, $\langle 1,0,0,0 \rangle$, $\langle 1,1,0,0 \rangle$, $\langle 1,1,1,0 \rangle$, $\langle 1,1,1,1 \rangle$, $\langle 0,1,1,1 \rangle$, $\langle 0,0,1,1 \rangle$, and $\langle 0,0,0,1 \rangle$.

A Heuristic for Adding Recovery. In the presence of faults, the program TR may reach states where there exists at least a process P_i $(0 \leq i \leq 3)$ whose x_i is corrupted (i.e., $x_i = \perp$). In such cases, processes P_i and $P_{((i+1)\ mod\ 4)}$ cannot take any transition, and as a result, the propagation of the token stops (i.e., the whole program deadlocks).

In order to recover from the states where there exist some corrupted processes, we apply the heuristic for single-step recovery from [2] in an iterative fashion. Specifically, we identify states from where single-step recovery to a set of states *RecoverySet* is possible. The initial value of *RecoverySet* is equal to the program invariant. At each iteration, we include a set of states in *RecoverySet* from where single-step recovery to *RecoverySet* is possible.

In the first iteration, we search for deadlock states where there is only one corrupted process in the ring. For example, consider a state $s_0 = \langle 1, \perp, 1, 0\rangle$. In the state s_0, P_1 and P_2 cannot take any transitions. However, P_3 can copy the value of x_2 and reach $s_2 = \langle 1, \perp, 1, 1\rangle$. Subsequently, P_0 changes x_0 to 0, and as a result, the program reaches state $s_3 = \langle 0, \perp, 1, 1\rangle$. The state s_3 is a deadlock state since no process can take any transition at s_3. To add recovery from s_3, we allow P_1 to correct itself by copying the value of x_0, which is equal to 0. Thus, by copying the value of x_0, P_1 adds a recovery transition to an invariant state $\langle 0, 0, 1, 1\rangle$. Therefore, we include s_3 in the set of states *RecoverySet* in the first iteration. Note that this recovery transition is added in low atomicity in that all the transitions included in action $(x_0 = 0) \wedge (x_1 = \perp) \rightarrow x_1 := 0$ can be included in the fault-tolerant program without violating safety.

In the second and third iterations, we follow the same approach and add recovery from states where there are two or three corrupted processes to states that we have already resolved in the previous iterations. Adding recovery up to the fourth iteration of our heuristic results in the intermediate program ITR $(1 \leq i \leq 3)$.

$$\begin{array}{lll}
ITR_0: & ((x_0 = 1) \vee (x_0 = \perp)) \ \wedge \ (x_3 = 1) & \longrightarrow x_0 := 0; \\
ITR_0': & ((x_0 = 0) \vee (x_0 = \perp)) \ \wedge \ (x_3 = 0) & \longrightarrow x_0 := 1; \\
ITR_i: & ((x_i = 0) \vee (x_i = \perp)) \ \wedge \ (x_{i-1} = 1) & \longrightarrow x_i := 1; \\
ITR_i': & ((x_i = 1) \vee (x_i = \perp)) \ \wedge \ (x_{i-1} = 0) & \longrightarrow x_i := 0;
\end{array}$$

Using above heuristic, we can only add recovery from the states where there exists at least one uncorrupted process. If there exists at least one uncorrupted process P_j $(0 \leq j \leq 3)$ then $P_{((j+1) \ mod \ 4)}$ will initiate the token circulation throughout the ring, and as a result, the program recovers to its invariant. However, in the fourth iteration of the above heuristic, we reach a point where we need to add recovery from the state where all processes are corrupted; i.e., $s_d = \langle \perp, \perp, \perp, \perp\rangle$. In such a state, the program ITR deadlocks as an action of the form $(x_0 = \perp) \wedge (x_1 = \perp) \rightarrow x_1 := 0$ cannot be included in the fault-tolerant program. Such an action can violate safety if x_2 and x_3 are not corrupted. In fact, no process can add safe recovery from s_d in low atomicity. Thus, *Add_Recovery* returns false for $\langle \perp, \perp, \perp, \perp\rangle$.

Adding the Actions of the High Atomicity Pseudo Process. In order to add masking fault-tolerance to the program ITR, a process P_{index} $(0 \leq index \leq 3)$ should set its x value to 0 (respectively, 1) when all processes are corrupted. Hence, we follow our synthesis method (cf. Figure 1), where the pseudo process PS_0 takes the high atomicity action HTR and recovers from s_d. Thus, the actions of the masking program MTR are as follows $(1 \leq i \leq 3)$.

$$\begin{array}{lll}
MTR_0: & ((x_0 = 1) \vee (x_0 = \perp)) \ \wedge \ (x_3 = 1) & \longrightarrow x_0 := 0; \\
MTR_0': & ((x_0 = 0) \vee (x_0 = \perp)) \ \wedge \ (x_3 = 0) & \longrightarrow x_0 := 1; \\
MTR_i: & ((x_i = 0) \vee (x_i = \perp)) \ \wedge \ (x_{i-1} = 1) & \longrightarrow x_i := 1; \\
MTR_i': & ((x_i = 1) \vee (x_i = \perp)) \ \wedge \ (x_{i-1} = 0) & \longrightarrow x_i := 0; \\
HTR: & (x_0 = \perp) \ \wedge \ (x_1 = \perp) \ \wedge \ (x_2 = \perp) \ \wedge \ (x_3 = \perp) & \longrightarrow x_0 := 0;
\end{array}$$

In order to refine the high atomicity action HTR, we need to add a detector that detects the state predicate $(x_0 = \bot) \wedge\ (x_1 = \bot) \wedge\ (x_2 = \bot) \wedge\ (x_3 = \bot)$. In Section 5, we describe the specification of fault-tolerance components, and we show how we use a distributed detector to refine high atomicity actions.

[illegible] Had we non-deterministically chosen to use PS_i $(i \neq 0)$ as the process that adds the high atomicity recovery action then the high atomicity action HTR would have been different in that HTR would write x_i. (We refer the reader to [13] for a discussion about this issue.)

5 Specifying Pre-synthesized Components

In this section, we describe the specification of fault-tolerance components (i.e., detectors and correctors). Specifically, we concentrate on detectors and we consider a special subclass of correctors where a corrector consists of a detector and a write action on the local variables of a single process.

5.1 The Specification of Detectors

We recall the specification of a detector component presented in [14,10]. Towards this end, we describe detection predicates, and witness predicates. A detector, say d, identifies whether or not a global state predicate, X, holds. The global state predicate X is called a [illegible] predicate in the global state space of a distributed program [14, 10].

It is often difficult to evaluate the truth value of X in an atomic action. Thus, we (i) decompose the detection predicate X into a set of smaller detection predicates $X_0 \cdots X_n$ where the compositional detection of $X_0 \cdots X_n$ leads us to the detection of X, and (ii) provide a state predicate, say Z, whose value leads the detector to the conclusion that X holds. Since when Z becomes true its value witnesses that X is true, we call Z a [illegible] predicate. If Z holds then X will have to hold as well. If X holds then Z will eventually hold and continuously remain [illegible]. Hence, corresponding to each detection predicate X_i, we identify a witness predicate Z_i such that if Z_i is [illegible] then X_i will be [illegible].

The detection predicate X is either the conjunction of X_i $(0 \leq i \leq n)$ or the disjunction of X_i. Since the detection predicates that we encounter represent deadlock states, they are inherently in conjunctive form where each conjunct represents the valuation to program variables at some process. Hence, in the rest of the paper, we consider the case where X is a conjunction of X_i, for $0 \leq i \leq n$.

Specification. Let X and Z be state predicates. Let 'Z detects X' be the problem specification. Then, 'Z detects X' stipulates that

- ([illegible]) When Z holds, X must hold as well.
- ([illegible]) When the predicate X holds and continuously remains [illegible], Z will eventually hold and continuously remain [illegible]. □

We represent the safety specification of a detector as a set of transitions that a detector is not allowed to take. Thus, the following set of transitions represents the safety specification of a detector.

$$spec_d = \{(s_0, s_1) : (Z(s_1) \ \wedge \ \neg X(s_1))\}$$

The predicate $Z(s_1)$ denotes the truth value of Z at state s_1.

5.2 The Representation of Detectors

In this section, we describe how we formally represent a distributed detector. While our method allows one to use detectors of different topologies (cf. Section 6.1), in this section, we comprehensively describe the representation of a linear (sequential) detector as such a detector will be used in our token ring example.

The Composition of Detectors. A detector, say d, with the detection predicate $X \equiv X_0 \wedge \ldots \wedge X_n$ is obtained by composing d_i, $0 \leq i \leq n$, where d_i is responsible for the detection of X_i using a witness predicate Z_i $(0 \leq i \leq n)$. The elements of d can execute in parallel or in sequence. More specifically, parallel detection of X requires $d_0 \cdots d_n$ to execute concurrently. As a result, the state predicate $(Z_0 \wedge \cdots \wedge Z_n)$ is the witness predicate for detecting X.

A sequential detector requires the detectors $d_0 \cdots d_n$ to execute one after another. For example, given a linear arrangement $d_n \cdots d_0$, a detector d_i $(0 \leq i < n)$ detects its detection predicate, using Z_i, after d_{i+1} witnesses. Thus, when Z_i becomes , it shows that Z_{i+1} already holds. Since when Z_i becomes X_i must be also , it follows that the detection predicates $X_n \cdots X_i$ hold. Therefore, we can atomically check the witness predicate Z_0 in order to identify whether or not $X \equiv (X_n \wedge \cdots \wedge X_0)$ holds.

The detection of global state predicates of programs that have a hierarchical topology (e.g., tree-like structures) requires parallel and sequential detectors. For brevity, we demonstrate our method in the context of a linear detector. As such a detector suffices for the example considered in this paper, we refer the reader to [7] for an illustration of this method for hierarchical components.

A Linear Detector. We consider a detector d with linear topology. The detector d consists of $n+1$ elements $(n > 0)$, its specification $spec_d$, its variables, and its invariant U. Since the structure of the detector is linear, without loss of generality, we consider an arrangement $d_n \cdots d_0$ for the elements of the distributed detector, where the left-most element is d_n and the right-most element is d_0.

Component Variables. Each element d_i, $0 \leq i \leq n$, of the detector has a Boolean variable y_i.

Read/Write Restrictions. Element d_i can read y_i and y_{i+1}, and can only write y_i $(0 \leq i < n)$. d_n reads and writes y_n. Also, d_i is allowed to read r_i; i.e., the set of variables that are readable for a process P_i with which d_i is composed.
Witness Predicates. The witness predicate of each d_i, say Z_i, is equal to $(y_i = true)$.

The Detector Actions. The actions of the linear detector are as follows $(0 \leq i < n)$.

$$DA_n : \ (LC_n) \ \wedge \ (y_n = false) \qquad \longrightarrow \qquad y_n := true;$$
$$DA_i : \ (LC_i) \ \wedge \ (y_i = false) \ \wedge \ (y_{i+1} = true) \qquad \longrightarrow \qquad y_i := true;$$

Using action DA_i $(0 \leq i < n)$, each element d_i of the linear detector witnesses (i.e., sets the value of y_i to [illegible]) whenever (i) the condition LC_i becomes [illegible], where LC_i represents a local condition that d_i atomically checks (by reading the variables of P_i), and (ii) its neighbor d_{i+1} has already witnessed. The detector d_n witnesses (using action DA_n) when LC_n becomes true.

Detection Predicates. The detection predicate X_i for element d_i is equal to $(LC_n \ \wedge \ \cdots \ \wedge \ LC_i)$ $(0 \leq i \leq n)$. Therefore, d_0 detects the global detection predicate $LC_n \wedge \cdots \wedge LC_0$.

Invariant. During the detection, when an element d_i sets y_i to true, the elements d_j, for $i < j \leq n$, have already set their y values to true. Hence, we represent the invariant of the linear detector by the predicate U, where

$$U = \{s : (\forall i : (0 \leq i \leq n) : (y_i(s) \Rightarrow (\forall j : (0 \leq j \leq n) \wedge (j > i) : LC_j))\}$$

Faults. We model the fault transitions that affect the linear detector using the following action (cf. Section 7 for a discussion about the way that we have modeled the faults).

$$F : true \longrightarrow y_i := false;$$

Theorem 5.1 The linear detector is masking F-tolerant for 'Z detects X' from U. (cf. [13] for proof.) □

5.3 Token Ring Example Continued

In Section 4.2, we added the following high atomicity action to the token ring program ITR that is executed by the pseudo process PS_0.

$$HTR: \ \ (x_0 = \bot) \ \wedge \ (x_1 = \bot) \ \wedge \ (x_2 = \bot) \ \wedge \ (x_3 = \bot) \ \ \longrightarrow \qquad x_0 := 0$$

In order to synthesize a distributed program (that includes low atomicity actions), we need to refine the guard of the above action. The read/write restrictions of the processes in the token ring program identify the underlying communication topology of the fault-intolerant program, which is a ring. Hence, we select a linear detector, d, so that we can organize its elements, d_3, d_2, d_1, d_0, in the ring. Each detector d_i is responsible to detect whether or not the local conditions LC_3 to LC_i hold $(LC_i \equiv (x_i = \bot))$, for $0 \leq i \leq 3$. Thus, the detection predicate X_i is equal to $((x_3 = \bot) \wedge \cdots \wedge (x_i = \bot))$, for $0 \leq i \leq 3$. As a result, the global detection predicate of the linear detector is $((x_3 = \bot) \wedge (x_2 = \bot) \wedge (x_1 = \bot) \wedge (x_0 = \bot))$. The witness predicate of each d_i, say Z_i, is equal to $(y_i = true)$, and the actions of the sequential detector are as follows $(0 \leq i \leq 2)$.

$$DA_3 : \ (x_3 = \bot) \ \wedge \ (y_3 = false) \qquad \longrightarrow \qquad y_3 := true;$$
$$DA_i : \ (x_i = \bot) \ \wedge \ (y_i = false) \ \wedge \ (y_{i+1} = true) \qquad \longrightarrow \qquad y_i := true;$$

Note that we replace (LC_i) with $(x_i = \bot)$ in the above actions. During the synthesis, after the synthesis algorithm acquires the actions of its required component, it replaces each (LC_i) with the appropriate condition in order to create the transition groups corresponding to each action of the component.

6 Using Pre-synthesized Components

In this section, we describe how we perform the second and the third step of our synthesis method presented in Figure 1. In particular, in Section 6.1, we show how we automatically specify the required components during the synthesis. Then, in Section 6.3, we show how we ensure that no interference exists between the program and the fault-tolerance component. Afterwards, we present an algorithm for the addition of fault-tolerance components. In Sections 6.2 and 6.4, we respectively present the algorithmic specification and the algorithmic addition of a linear detector to the token ring program.

6.1 Algorithmic Specification of the Fault-Tolerance Components

We present the Component_Specification algorithm (cf. Figure 2) that takes a deadlock state s_d, the distribution restrictions (i.e., the read/write restrictions) of the program being synthesized, and the set of high atomicity pseudo processes PS_i $(0 \leq i \leq n)$. First, the algorithm searches for a high atomicity process PS_{index} that is able to add a high atomicity recovery action, $ac : grd \rightarrow st$, from s_d to a state in the state predicate S_{rec}, where S_{rec} represents the set of states from where there exists a safe recovery path to the invariant. Also, we verify the closure of $S_{rec} \cup s_d$ in the computations of $p[]f$. If there exists such a process PS_{index} then the algorithm returns a triple $\langle X, R, index\rangle$, where (i) X is the detection predicate that should be refined in the refinement of the action ac; (ii) R is a relation that represents the topology of the program, and (iii) the $index$ is an integer that identifies the process that should detect grd and execute st.

The Component_Specification algorithm constructs the state predicate X using the LC_i conditions. Each LC_i condition is by itself a conjunction that consists of the program variables readable for process P_i. Therefore, the predicate X will be the conjunction of LC_i conditions $(0 \leq i \leq n)$.

The relation $R \subseteq (P \times P)$ identifies the communication topology of the distributed program, where P is the set of program processes. We represent R by a finite set $\{\langle i, j\rangle : (0 \leq i \leq n) \wedge (0 \leq j \leq n) : w_i \subseteq r_j\}$ that we create using the read/write restrictions among the processes. The presence of a pair $\langle i, j\rangle$ in R shows that there exists a communication link between P_i and P_j. Since we internally represent R by an undirected graph, we consider the pair $\langle i, j\rangle$ as an unordered pair.

The Interface of the Fault-Tolerance Components. The format of the interface of each component is the same as the output of the Component_Specification algorithm, which is a triple $\langle X, R, index\rangle$ as described above. We use this interface to extract a component from the component library using a pattern-matching algorithm. Towards this end, we use existing specification-matching techniques [15]. For reasons of space, we omit the details of the component extraction from the library of components.

The Output of the Component Library. Given the interface $\langle X, R, index\rangle$ of a required component, the component library returns the witness predicate,

```
Component_Specification($s_d$: state, $S_{rec}$: state predicate, $PS_0, \cdots, PS_n$: high atomicity
                    pseudo process, $spec$: safety specification, $r_0, \cdots, r_n$: read restrictions,
                                                              $w_0, \cdots, w_n$: write restrictions)
{ // $n$ is the number of processes.
  if ( $\exists index : 0 \leq index \leq n : (\exists s : s \in S_{rec} : (s_d, s) \in PS_{index} \wedge$
                    $((s_d, s)$ does not violate $spec) \wedge (\forall x : (x(s_d) \neq x(s)) : x \in w_{index}))$ )
  then      $X := \wedge_{i=0}^{n}(LC_i)$, where $LC_i = (\wedge^{|r_i|}(x = x(s_d)))$;
            $R = \{\langle i, j \rangle : (0 \leq i \leq n) \wedge (0 \leq j \leq n) : w_i \subseteq r_j\}$;
            return $X$, $R$, $index$;
  else return $false$, $\emptyset$, -1;
}
```

Fig. 2. Automatic specification of a component

Z, the invariant, U, and the set of transition groups, $gd_0 \cup \cdots \cup gd_k \cup g_{index}$, of the pre-synthesized component ($k \geq 0$). The group of transitions g_{index} represents the low atomicity write action that should be executed by process P_{index}.

Complexity. Since the algorithm Component_Specification checks the possibility of adding a high atomicity recovery action to each state of S_{rec}, its complexity is polynomial in the number of states of S_{rec}.

6.2 Token Ring Example Continued

We trace the algorithm of Figure 2 for the case of token ring program. First, we non-deterministically identify PS_0 as the process that can read every program variable and can add a high atomicity recovery transition from the deadlock state $s_d = \langle \perp, \perp, \perp, \perp \rangle$. Thus, the value of *index* will be equal to 0. Second, we construct the detection predicate X, where $X \equiv ((x_0 = \perp) \wedge (x_1 = \perp) \wedge (x_2 = \perp) \wedge (x_3 = \perp))$. Finally, using the read/write restrictions of the processes in the token ring program, the relation R will be equal to $\{\langle 0, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 0 \rangle\}$.

6.3 Algorithmic Addition of The Fault-Tolerance Components

In this section, we present an algorithm for adding a fault-tolerance component to a fault-intolerant distributed program to resolve a deadlock state s_d. Before the addition, we ensure that no interference exists between the program and the fault-tolerance component; i.e., the execution of one of them does not violate the (safety or liveness) specification of the other one. We show that our addition algorithm is sound; i.e., the synthesized program satisfies the requirement of the addition problem (cf. Section 3).

We represent the transitions of p by the union of its groups of transitions (i.e., $\cup_{i=0}^{m} g_i$). We also assume that we have extracted the required pre-synthesized component, c, as described in Section 6.1. The component c consists of a detector d that includes a set of transition groups $\cup_{i=0}^{k} gd_i$, and the write action of the pseudo process PS_{index} represented by a group of transitions g_{index} in the low atomicity.

The state space of the composition of p and d is the new state space $S_{p'}$. We introduce an onto function $H_1 : S_{p'} \rightarrow S_p$ (respectively, $H_2 : S_{p'} \rightarrow S_d$) that maps the states in the new state space $S_{p'}$ to the states in the old state space S_p (respectively, S_d where S_d is the state space of the detector d). Now, we show how we verify the interference-freedom of the composition of c and p.

Interference-Freedom. In order to ensure that no interference exists between p and c, we have to ensure the following three conditions hold in the new state space $S_{p'}$: (i) transitions of p do not interfere with the execution of d; (ii) transitions of d do not interfere with the execution of p, and (iii) the low atomicity write action associated with c does not interfere with the execution of p and d.

First, we ensure that the set of transitions of p do not interfere with the execution of d by constructing the set of groups of transitions I_1, where I_1 contains those groups of transitions in the new state space $S_{p'}$ that violate either the safety of d or the closure of its invariant U.

$$\begin{aligned} I_1 = \{g : (\exists g_j : (g_j \in p) \wedge (0 \leq j \leq m) : (H_1(g) = g_j) \wedge \\ (\exists (s'_0, s'_1) : (s'_0, s'_1) \in g : ((s'_0, s'_1) \ \text{ violates } \ spec_d) \vee \\ (H_2(s'_0) \in U \ \wedge \ H_2(s'_1) \notin U))\} \end{aligned}$$

The transitions of p do not interfere with the liveness of d because d need not execute when p is not in the deadlock state s_d. Thus, p does not affect the liveness of d. Hence, we are only concerned with the safety of the detector d and the closure of U. When we map the transitions of p to the new state space, the mapped transitions should preserve the safety of d. Moreover, if the image of a transition (s'_0, s'_1) starts in U (i.e., $H_2(s'_0) \in U$) then the image of (s'_0, s'_1) will have to end in U (i.e., $H_2(s'_1) \in U$). The emptiness of I_1 shows that the transitions of p do not interfere with the execution of d.

Likewise, we construct the set of groups of transitions I_2 and I_3 in the new state space $S_{p'}$ to verify the second and the third conditions of interference-freedom. Since I_2 and I_3 are structurally similar to I_1, we skip their presentation (cf. [13] for details). Thus, if I_1, I_2, and I_3 are empty then we declare that no interference will happen due to the addition of c to p.

Addition. We present the Add_Component algorithm for an interference-free addition of the fault-tolerance component c to p. In the new state space $S_{p'}$, we construct a set of transition groups p_{H_1} (respectively, d_{H_2}) that includes all groups of transitions, g, whose images in S_p (respectively, S_d) belong to p (respectively, d). Besides, no transition of $(s'_0, s'_1) \in g$ violates the safety specification of d (respectively, p) or the closure of the invariant of d (respectively, p), i.e., U (respectively, S). Note that in the set d_{H_2}, the image of every group g in d and p must belong to the same process.

The set p_c includes all groups of transitions, g, whose every transition has an image in g_{index} under the mapping H_2. Further, no transition $(s'_0, s'_1) \in g$ violates the safety of *spec* or the closure of S.

The set of states of the invariant of the synthesized program, S', consists of those states whose images in S_p belong to the program invariant S and whose images in the state space of the detector, S_d, belong to the detector invariant U.

```
Add_Component($S, S_{rec}, U$: state predicate, $H_1, H_2$: onto mapping function,
                    $spec, spec_d$: safety specification, $g_0, \cdots, g_m$, $gd_0, \cdots, gd_k, g_{index}$: groups of transitions)
{ // $p = g_0 \cup \cdots \cup g_m$, and $d = gd_0 \cup \cdots \cup gd_k \cup g_{index}$
// $P_0 \cdots P_n$ are the processes of $p$, and $d_0 \cdots d_n$ are the elements of $d$

  $p_{H_1} = \{g : (\exists g_j : (g_j \in p) \wedge (0 \leq j \leq m) : (H_1(g) = g_j) \wedge$
         $(\forall (s'_0, s'_1) : (s'_0, s'_1) \in g : ((s'_0, s'_1)$ does not violate $spec_d) \wedge (H_2(s'_0) \in U \Rightarrow H_2(s'_1) \in U))\}$

  $d_{H_2} = \{gd : (\exists gd_j : (gd_j \in d) \wedge (0 \leq j \leq k) : (H_2(gd) = gd_j) \wedge$
         $(\exists d_i, P_l : (0 \leq i \leq n) \wedge (0 \leq l \leq n) : (H_2(gd) \in d_i) \wedge (H_1(gd) \in P_l) \wedge (l = i)) \wedge$
         $(\forall (s'_0, s'_1) : (s'_0, s'_1) \in gd : ((s'_0, s'_1)$ does not violate $spec) \wedge (H_1(s'_0) \in S \Rightarrow H_1(s'_1) \in S))\}$

  $p_c := \{g : (H_2(g) = g_{index}) \wedge (\forall (s'_0, s'_1) : (s'_0, s'_1) \in g : ((s'_0, s'_1)$ does not violate $spec) \wedge$
              $(H_1(s'_1) \in S_{rec}) \wedge (H_2(s'_0) \in U \Rightarrow H_2(s'_1) \in U) \wedge ((s'_0, s'_1)$ does not violate $spec_d))\}$
  $S' := \{s : s \in S_{p'} : H_1(s) \in S \wedge H_2(s) \in U\}$
  $p' := p_{H_1} \cup d_{H_2} \cup p_c$;
  return $p', S'$;
}
```

Fig. 3. The automatic addition of a component

Theorem 6.1 The algorithm Add_Component is sound. (cf. [13] for proof.) □

Theorem 6.2 The complexity of Add_Component is polynomial in $S_{p'}$. (cf. [13] for proof.) □

6.4 Token Ring Example Continued

Using Add_Component, we add the detector specified in Section 6.2 to the token ring program MTR introduced in Section 4.2. The resulting program, consisting of the processes $P_0 \cdots P_3$ arranged in a ring, is masking fault-tolerant to process-restart faults. We represent the transitions of P_0 by the following actions.

MTR_0 :	$((x_0 = 1) \vee (x_0 = \perp)) \wedge (x_3 = 1)$	$\longrightarrow x_0 := 0;$
MTR'_0 :	$((x_0 = 0) \vee (x_0 = \perp)) \wedge (x_3 = 0)$	$\longrightarrow x_0 := 1;$
D_0 :	$(x_0 = \perp) \wedge (y_0 = false) \wedge (y_1 = true)$	$\longrightarrow y_0 := true;$
C_0 :	$(y_0 = true)$	$\longrightarrow x_0 := 0;\ y_0 := false;$

The actions MTR_0 and MTR'_0 are the same as the actions of the MTR program presented in Section 4.2. The action D_0 belongs to the sequential detector that sets the witness predicate Z_0 to true. The action C_0 is the recovery action that P_0 executes whenever the witness predicate $(y_0 = true)$ becomes [illegible]. Now, we present the actions of P_3.

MTR_3 :	$((x_3 = 0) \vee (x_3 = \perp)) \wedge (x_2 = 1)$	$\longrightarrow x_3 := 1; y_3 := false;$
MTR'_3 :	$((x_3 = 1) \vee (x_3 = \perp)) \wedge (x_2 = 0)$	$\longrightarrow x_3 := 0; y_3 := false;$
$D3$:	$(x_3 = \perp) \wedge (y_3 = false)$	$\longrightarrow y_3 := true;$

The action D_3 belongs to the detector d_3 that sets Z_3 to [illegible]. We represent the transitions of P_1 and P_2 as the following parameterized actions (for $i = 1, 2$).

MTR_i :	$((x_i = 0) \vee (x_i = \perp)) \wedge (x_{i-1} = 1)$	$\longrightarrow x_i := 1; y_i := false;$
MTR'_i :	$((x_i = 1) \vee (x_i = \perp)) \wedge (x_{i-1} = 0)$	$\longrightarrow x_i := 0; y_i := false;$
D_i :	$(x_i = \perp) \wedge (y_i = false) \wedge (y_{i+1} = true)$	$\longrightarrow y_i := true;$

The above program is masking fault-tolerant for the faults that corrupt one or more processes. Note that when a process P_i $(1 \leq i \leq 3)$ changes the value of x_i to a non-corrupted value, it falsifies Z_i (i.e., y_i). The falsification of Z_i is important during the recovery from $s_d = \langle \perp, \perp, \perp, \perp \rangle$ in that when x_i takes a non-corrupted value, the detection predicate X_i no longer holds. Thus, if Z_i remains true then the detector d_i witnesses incorrectly, and as a result, violates the safety of the detector. However, P_0 does not need to falsify its witness predicate Z_0 in actions MTR_0 and MTR'_0 because the action C_0 has already falsified Z_0 during a recovery from s_d.

7 Discussion

In this section, we address some of the questions raised by our synthesis method. Specifically, we discuss the following issues: the model of faults considered in this paper, the fault-tolerance of the components, the choice of detectors and correctors, and pre-synthesized components with non-linear topologies.

[illegible]

Yes. The notion of state perturbation is general enough to model different types of faults (namely, stuck-at, crash, fail-stop, omission, timing, or Byzantine) with different natures (intermittent, transient, and permanent faults). As an illustration, we modeled the process-restart faults that affect the token ring program, presented in this paper, by state perturbation. This model has also been used in designing fault-tolerance to (i) fail-stop, omission faults (e.g., [9]), (ii) transient faults and improper initialization (e.g., [16]), and (iii) input corruption (e.g., [9]).

[illegible]

Yes. The added component may itself be perturbed by the fault to which fault-tolerance is added. Hence, the added component must itself be fault-tolerant. For example, in our token ring program, we modeled the effect of the process restart on the added component and ensured that the component is fault-tolerant to that fault (cf. Theorem 5.1). For the fault-classes that are commonly used, e.g., process failure, process restart, input corruption, Byzantine faults, such modeling is always possible. For arbitrary fault-classes, however, some [illegible] may be required to ensure that the modeling is appropriate for that fault.

[illegible]

While there are several approaches (e.g., [17]) that manually transform a fault-intolerant program into a fault-tolerant program, we use detectors and correctors in this paper, based on their necessity and sufficiency for manual addition of fault-tolerance [5]. The authors of [5] have also shown that detectors and correctors are abstract enough to generalize other components (e.g., comparators and voters used in replication-based approaches) for the design of fault-tolerant programs. Hence, we expect that the synthesis method in this paper can benefit from the generality of detectors and correctors in the [illegible] synthesis of fault-tolerant programs as there is a potential to provide a rich library of fault-tolerance components. Moreover, pre-synthesized detectors provide the kind of abstraction by which we can integrate efficient existing detections approaches (e.g., [18, 19]) in pre-synthesized fault-tolerance components.

[illegible]

Yes. Using the synthesis method of this paper, we have added presynthesized components with tree-like structure to a diffusing computation program [7] where we synthesize a program that is fault-tolerant to the faults that perturb the state of the diffusing computation and the program topology.

8 Conclusion and Future Work

In this paper, we identified an approach for the synthesis of fault-tolerant programs from their fault-intolerant versions using pre-synthesized fault-tolerance components. Our approach differs from the synthesis method presented in [20] where one has to synthesize a fault-tolerant program from its temporal logic specification. Specifically, we demonstrated a hybrid synthesis method that combines heuristics presented in [2, 3, 4] with pre-synthesized detectors and correctors. We presented a sound algorithm for automatic specification and addition of pre-synthesized detectors/correctors to a distributed program. We showed how one could verify the interference-freedom of the fault-intolerant program and the added components. Using our synthesis algorithm, we showed how masking fault-tolerance is added to a token-ring program where all processes may be corrupted. By contrast, the previous synthesis algorithms fail to synthesize a fault-tolerant program when all processes are corrupted.

We also extended the problem of adding fault-tolerance to the case where new variables can be introduced while synthesizing fault-tolerant programs. By contrast, previous algorithms required that the state space of the fault-tolerant program is the same as that of the fault-intolerant program. Moreover, our synthesis method controls the way new variables are introduced; new variables are determined based on the added components. Hence, the synthesis method controls the way in which the state space is expanded.

References

1. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, page 82, 2000.
2. S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of byzantine agreement. *Symposium on Reliable Distributed Systems*, 2001.
3. S. S. Kulkarni and A. Ebnenasir. The complexity of adding failsafe fault-tolerance. *International Conference on Distributed Computing Systems*, pages 337–334, 2002.
4. S. S. Kulkarni and A. Ebnenasir. Enhancing the fault-tolerance of nonmasking programs. *International Conference on Distributed Computing Systems*, pages 441–450, 2003.

5. S. S. Kulkarni. *Component-based design of fault-tolerance.* PhD thesis, Ohio State University, 1999.
6. Ali Ebnenasir and Sandeep S. Kulkarni. FTSyn: A framework for automatic synthesis of fault-tolerance. `http://www.cse.msu.edu/~ebnenasi/research/tools/ftsyn.htm`.
7. Ali Ebnenasir and S.S. Kulkarni. Hierarchical presynthesized components for automatic addition of fault-tolerance: A case study. *In the extended abstracts of the ACM workshop on the Specification and Verification of Component-Based Systems (SAVCBS), Newport Beach, California*, 2004.
8. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
9. A. Arora and M. G. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.
10. A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *International Conference on Distributed Computing Systems*, pages 436–443, May 1998.
11. P. Attie and A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM TOPLAS (a preliminary version of this paper appeared in PODC96)*, 23(2), March 2001.
12. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1990.
13. S. S. Kulkarni and Ali Ebnenasir. Adding fault-tolerance using presynthesized components. *Technical report MSU-CSE-03-28, Department of Computer Science, Michigan State University, East Lansing, Michigan, USA. A revised version is available at http://www.cse.msu.edu/~sandeep/auto_component_techreport.ps*, 2003.
14. A. Arora and S. S. Kulkarni. Component based design of multi-tolerant systems. *IEEE Transactions on Software Engineering*, 1998.
15. A. Moormann Zaremski and J.M. Wing. Specification matching of software components. *in proceedings of the 3 rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.
16. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 1974.
17. Z. Liu and M. Joseph. Transformations of programs for fault-tolerance. *Formal Aspects of Computing*, 1992.
18. A.I. Tomlinson and V.K. Garg. Detecting relational global predicates in distributed systems. *In proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, California.*, pages 21–31, May 1993.
19. Neeraj Mittal. *Techniques for Analyzing Distributed Computations.* PhD thesis, The University of Texas at Austin, 2002.
20. A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC)*, 1998.

Efficiency of Dynamic Arbitration in TDMA Protocols

Jens Chr. Lisner

University of Duisburg-Essen, D-45117 Essen, Germany
lisner@informatik.uni-essen.de

Abstract. Several existing TDMA-based solutions, which allow dynamic allocation of slots, are using minislotting strategies. Because of this approach they tend to loose bandwidth under some circumstances. Moreover, they cannot guarantee fault tolerance together with dynamic allocation. In [13] the *Tea*protocol was introduced, which achieves both flexibility and fault tolerance of a TDMA-based protocol using dynamic media access. This paper shows, that the extension method of *Tea*is also an efficient solution compared to two other arbitration strategies which are implemented in real-world protocols. The agreement-based media allocation in *Tea*exhibits better utilization of the extra bandwidth which is reserved for dynamic arbitration. Moreover, this paper presents an extension of *Tea*which allows for flexible message length which contributes further to efficiency.

1 Motivation

In the past few years different protocols for use in automotive real-time applications have been developed. Fault tolerance and realtime requirements induced the need for deterministic behavior of the protocol. This was realized by using static length slots with pre-configured scheduling which made it possible to use guardians for avoiding "babbling idiot" faults, and distributed fault-tolerant clock synchronization. These concepts are implemented in TTP/C and parts of FlexRay. On the other side there is also a need for dynamic media access. In the mixed-mode protocol FlexRay, there are separate segments with static slotting and dynamic minislotting, respectively.

The minislotting approach allows to send messages without fixed length, or sending nothing at all. The slot size is adjusted to the message length, or to minislot size if there is no media access. The drawback of this method is the waiting time for incoming transmission in every minislot between messages costs efficiency. The time needed for this process cannot be used, even if there is data to transmit. Moreover it is not possible to protect slots with dynamic length by guardians. A single faulty node can cause collisions on the bus.

The guardian approach protects the channels by denying access to the bus if the controller tries to send outside its slots. Therefore the guardian requires the same time-base as the controller. Moreover, for fault tolerance reasons the guardian's time-base should be independent from the controller's clock. Different solutions have been developed to overcome this problem. Every guardian could have an own clock, and its own clock synchronization. This is a nearly perfect solution for static schedules, but too expensive. Another popular solution requires a centralized guardian ([1]), which resides on a star coupler. A guardian fault (e.g. in the protocol node) therefore can make the whole channel useless, while a controller fault can be tolerated. A proposal is made in [5] to use a time base which is loosely based on the controllers clock. This is a cheap solution that makes it possible to detect the major timing problems. However, it cannot assure that minor changes cause errors over longer periods of time.

The [1] protocol as introduced in [13] overcomes this limitations. is a mixed-mode protocol like FlexRay, but exhibits substantially changes in dynamic arbitration. There is a static "regular" part, and a dynamically allocated "extension" part. In the regular part it is possible for each controller to request one slot for the extension part. Therefore the regular part itself is splitted into two phases of equal length: a request and a confirmation phase. In opposite to most time-triggered protocols, requires that two different controllers are sharing one time-slot on two different channels. This way, it is possible that every controller sends once on both phases. A controller sends on different channels in the request and extension phases. Controllers which want to send messages in the extension part can send a request in the request phase. The requests are piggybacked to normal messages. In the confirmation phase, the schedule for the extension part is build through majority voting. The controllers have to send a vector with the received requests (piggybacked to normal messages as well). At the end of the regular part the schedule for the extension part is known. It is not more necessary to wait for transmission in minislots.

is able to tolerate controller together with channel faults. Both faulty controller and faulty channel can behave arbitrarily at any time. This means in the case of a faulty controller, that it can send correctly coded or corrupted messages, or no messages at all. A faulty channel can corrupt messages, deliver messages only to a subset of all connected controllers, or no signals at all. Only the spontaneous generation of correctly coded messages is excluded.

To protect the channels, introduces a new node architecture with two independent controllers which guard each other (see Fig. 1). Each controller can grant or deny its neighbor access to the bus, by opening or closing the appropriate switches. Aside from the access control, both controllers are independent in function, and can be used by different hosts, or to build up a duplex-system as shown in Fig. 1. During runtime, the role of the controllers as guard and sender changes when following the current schedule.

The advantage is, that there are no timing problems with respect to differences in the distributed time base used by different controllers. This eliminates the need for synchronization with dumb components as used in architectures with local bus guardian components. It is also assured, that there is no direct

[1] *T*ime-triggered *e*fficiency by *a*greement

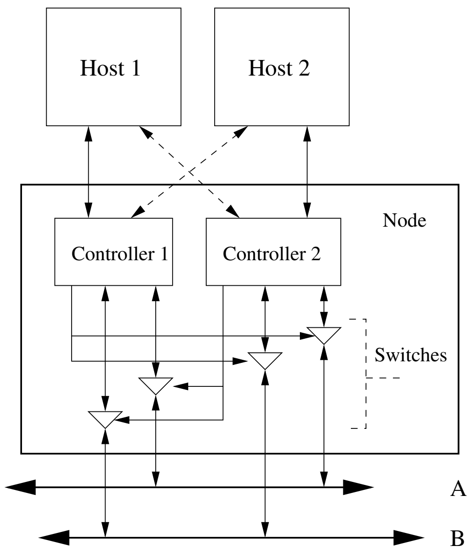


Fig. 1. Architecture of a *Tea*node. Two different communication controllers are guarding each other in one node. Each controller offers communication services for one host. As a side effect it is possible to build up a duplex-system, if the hosts can use both controllers (see dashed lines)

dependency between a controller and its neighbor. The time bases are depending both on the global time, which is shared by any fault-free controllers in the network. The global time base is normally the result of a fault-tolerant clock synchronization algorithm. As a result, timing faults of a neighbor controller are tolerated. The disadvantage is, that a faulty controller can deny a fault-free neighbor controller access to the channels at any time. In any case fail-silent behavior is guaranteed in the presence of timing faults.

This paper focusses on efficiency of different arbitration methods. First the known strategies with assigned slots as used in FlexRay and byteflight[2] and the agreement method in are presented. Example calculations will be given in section 3.

2 Strategies for Dynamic Media Access in TDMA-Based Protocols

This section discusses three different arbitration methods in a TDMA-context. For analysis, an accurate time model is developed in section 2.1. In section 2.2 the arbitration schemes are presented and the possible overhead is investigated for each method. Section 2.3 discusses the impact of possible channel and controller

[2] byteflight is a lightweight protocol without guardians and static segments. It is not designed under the aspect of fault-tolerance.

faults in the agreement-based method, which is the only fault-tolerant method of the methods presented here.

2.1 The Time Model

For all protocols there are restrictions through the limited precision of the clocks of the controllers. It is assumed that the values for the maximum clock rate deviation is given, by r_f for the fastest and r_s for the slowest controller.[3] Both should be the maximum values. Furthermore there may be an offset $o_{f,s}$ between two clocks f and s, which can be given in real time.

In TDMA-based protocols it must be assured that the distance between two events (transmission of messages) is large enough, that all fault-free controllers see them in the same slot. Otherwise there will be collisions. To get comparable results, the calculations below are based on the same time model, which puts a minimum constraint on timing parameters.

For the local clock time values the real time (e.g. measured in ns) can be calculated. The real time is given by $T_j\left(c\right) = \gamma_j\, c + o$, where c is the local time of controller j, and $\gamma_j = 1 + r_j$ the rate of its clock.

At a specific local time c, the difference in real time between the slowest controller s and the fastest controller f is $\Delta\left(c\right) = T_f\left(c\right) - T_s\left(c\right) = \left(\gamma_f - \gamma_s\right) c + o_{f,s}$, where $o_{f,s}$ is the initial offset between clocks f and s. Of importance is the maximum clock difference, because in time-triggered protocols it must be assured, that two clocks reached finally a local clock time c. Otherwise synchronous slot counting cannot be achieved. The maximum difference is reached at the end of the cycle just before the clocks are synchronized again. Given the length of the cycle is L, then $\Delta_{max} = \Delta(L)$. This value is used in a minislot to assure that all controllers are in the same minislot. The calculation of Δ_{max} is illustrated in Fig. 2.

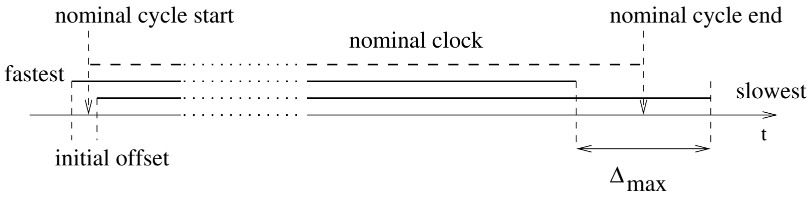


Fig. 2. The time model is based on the maximum allowed deviation between the clocks of the controllers. Assuming that clock synchronization is done at the end of the cycle, the distance between the slowest and the fastest clock Δ_{max} is highest

2.2 Media Access Methods

This section presents two media access methods. The first one assigns controllers to different slot of dynamic length at configuration time, while the method is

[3] In the following, the "clock i" should mean "clock of controller i."

based upon agreement during runtime. Since the minislotting approach with pre-assigned slots has no fault-tolerance capabilities, possible influences on efficiency for the prevention of faults are discussed in section 2.3.

Minislotting with Assigned Slots. If the order is pre-configured, but it is not known who will send, it is possible to count up short slots, until some controller starts to send in its minislot. After transmission has stopped, minislot accounting is continued until a further controller sends eventually.

The shortest possible slot, that is needed to wait for a possible transmission to start, is a minislot. In contrast to the priority based method, a minislot has constant length and must be pre-configured. Then a slot of dynamic length is inserted. It lasts for the duration of the sent message.

For calculation of the length of a minislot it must be taken into account that a sender must have enough time to start sending his packet before the fastest controller passes to the next minislot. A controller needs to assure, that all controllers passed to the next minislot. From this point on the controller has to wait until the slowest controller were able to send (see Fig. 3). This means that for a minislot length $\lambda_{minislot} > 2\,\Delta_{max}$ holds.

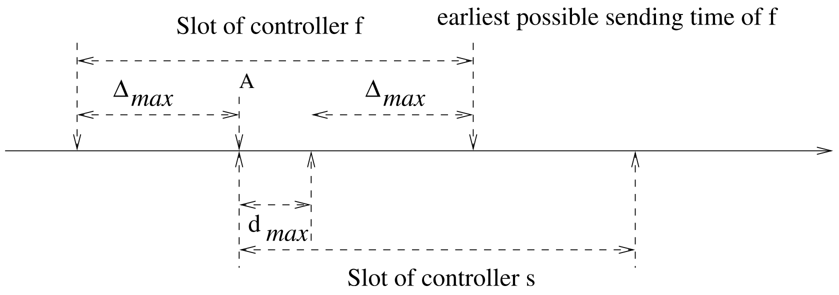


Fig. 3. Timing of a minislot. The controller f sends earliest at point A

The maximum delay caused by the transmission will be estimated by d_{max}. Additionally a small error ϵ will be taken into account, that subsumed possible effects such as discretization or rounding error through clock measurement or clock correction term calculation, that are not covered otherwise, and serves as a "safety margin". Finally the minislot length is $\lambda_{minislot} \geq 2\,\Delta_{max} + d_{max} + \epsilon$.

When a message is sent, the slot becomes a multiple of a minislot, which depend on the size of the message. A message cannot end in the same minislot it was started. Otherwise the transmission would not stop in the same slot, because the fastest controller passed to the next slot after the (possible) start of a transmission of the slowest one. The message must fit into a sequence of two or more minislots. If $\lambda_{message}(i)$ is the length of the message in real-time, then the transmission length can be estimated by $\lambda_{slot}(i) = \left(\left\lceil \frac{\lambda_{message}(i)}{\lambda_{minislot}} \right\rceil + 1\right) \lambda_{minislot} \geq 2\,\lambda_{minislot}$. The ceiling operator assures that the slot length is an integer multiple of the minislot length.

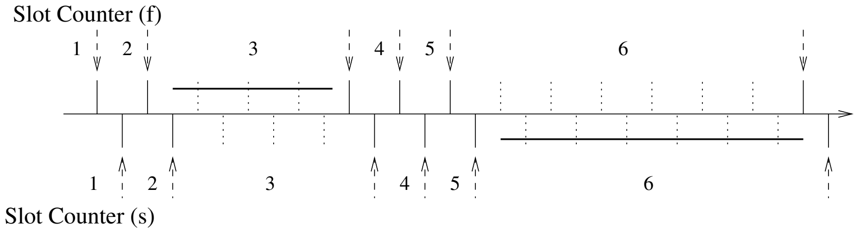


Fig. 4. Assigned slots: The fastest controller f is assigned to slot 3, the slowest s to slot 6. Slots 1,2 and 4, 5 are left unused

To calculate the amount of time T_{as} in minislots that gets lost by the arbitration method, let m_{last} be the last minislot of the cycle which is needed by a transmission. It must be taken into account, that a controller has to wait Δ_{max} until it is allowed to send. On the other side, there must be at least the same time available at the end of the message. Then, if k is the number of messages in the dynamic part, T_{as} is given by the following equation.

$$T_{as} = m_{last}\,\lambda_{minislot} - \sum_{i=1}^{k} \lambda_{slot}\,(i) + k\,2\,\Delta_{max} \tag{1}$$

As mentioned above, a controller can only start sending within a slot, if it is configured to do so. This schedule is important for the efficiency of a protocol. The consequence is that there can be gaps between two messages if a controller leaves the slot it is assigned to empty. These gaps are at least $\lambda_{minislot}$ long. Additionally, the order of controllers in the schedule works as some kind of priority. Controllers who are scheduled for higher slots may not be able to send, if the maximum cycle length is filled up with messages from other controllers.

Minislotting with Relaxed Timing. A more general method does not restrict the slot length to be an integer multiple of minislot length, if a controller is sending. The slot timing restarts after an idle state has been detected. After the end of the message has been received, the idle state can be detected by a timeout of duration $\delta_{final} + d_{max} + \epsilon$.

Thereafter each controller advances to the next minislot. Then the normal minislotting scheme is continued until a controller starts sending in its minislot. The overall overhead T_{ls} where k is the last index of the used slots and h is the number of used slots is given by

$$T_{ls} = (k-h)\,\lambda_{minislot} + h\,(\delta_{final} + d_{max} + \epsilon)\quad . \tag{2}$$

Agreement-Based Scheduling. If a schedule can be build up dynamically, before the dynamic part of the cycle starts, then the time between two messages reduces to a minimum. This done in the preceding regular part which has fixed scheduling by an agreement algorithm, as realized in the protocol in [13]

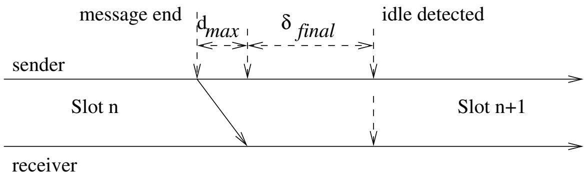


Fig. 5. Relaxed slot timing

(see section 1). The schedule contains only the sender who really wants to send. Consequently no special arbitration technique is necessary. The controller simply sent in the order of the prebuilt schedule.

In contrast to the variant presented in [13], the method described here should allow variable length messages. To achieve this a timeout δ_{final} for the detection of the end of the transmission is introduced. The value depends on the encoding of the messages, and is the time, the controller needs to detect an idle bus. Then the neighbor controller of the former sender must close the access to the bus, while the neighbor of the next sender has to grant the access. For both actions a small time interval d_{switch} has to be taken into account. Finally the necessary distance between two messages is given by $\lambda_{next} = \delta_{final} + d_{max} + d_{switch} + \epsilon$. At the beginning of the extension part or when the message receives maximum length, δ_{final} is not necessary. In this case the switches can be set immediately according to the schedule.

As tribute to the agreement algorithm, piggybacking of requests and request vectors (see 1) costs overhead in the regular part. This depends on the total number of controllers m, and the duration of a bit in real time d_{bit}. The overhead for the request phase is given by $m\,d_{bit}$. In the confirmation phase, each vector elements represents one of the states [illegible], [illegible] or [illegible], so that the vector would be $2m$ bits long. This vector can be packed, since the eight combinations of states of two controllers can be represented by three bits. Thus the overhead due to piggybacking in the regular part is given by $T_{reg} = m\,d_{bit} + 3\,\frac{m}{2}\,d_{bit}$.

For calculating the complete amount of time T_{ts} that is left unused for payload data, h is considered to be the total number of controllers sending in the extension, and k the number of controllers which do not utilize the maximum message length.

$$T_{ts} = \sum_{h} (d_{max} + d_{switch} + \epsilon) + \sum_{k} \delta_{final} + T_{reg} \qquad (3)$$

While the minislot approach needs at least a slot for detection of the beginning of a message, and the end of the message, the gaps between two messages can be significant smaller in agreement-based scheduling. The way how the senders are selected leaves many possibilities for scheduling including priorities, deadline scheduling, etc. This is not possible with pre-assigned schedules. The arbitration method enforces no restrictions.

2.3 Fault-Tolerance of Dynamic Media Access Methods

A novelty in the agreement-based approach is that the dynamic part can be implemented in a fault-tolerant way. This is achieved through the new node architecture as presented in section 1. It has a guaranteed fail-silent behavior, and can tolerate faults in the dynamic case, which is not possible with the classic guardian approach.

If the channels are known to be fault-free, then faults are seen by any controller the same way. On timeouts or errors during decoding on one of the channels, the reception can be stopped and all controllers can recalibrate the switches according to the schedule. This happens also if the maximum length of a message is reached. In the case when the controllers do not send at all, there is no additional cost, because the timeout mechanism shown in section 2.2 works as expected. Otherwise the time, left from start of sending until the reception is cancelled, has to be added. The prerequisite is that blocking causes a controller to detect either a CRC error, or sets the bus to an idle state.

In addition to a faulty controller one channel can be faulty. A faulty channel behaves arbitrary throughout the whole cycle. It could be reasonable to assume that a corruption of a message can be seen by any controller, so that the following condition holds: If one controller gets a correct message, then all others get a correct message, too. This is always the case if the sender is fault-free. Further, if a faulty, or blocked controller sends a corrupted message on the fault-free channel, all controllers see either a correct or a corrupted message (depending on what the controller did send on that channel). Under this circumstances it is possible to ignore the corrupted channel. The case of message corruption on both channels can be handled the same way as described above.

When a faulty channel acts asymmetrically, then two controllers could see two completely different behaviors on the channels. In Fig. 6, j sees a message from the faulty controller f on the faulty channel. Due to the asymmetric behavior of the faulty channel, k recognizes an idle channel. As f does not send on the fault-free channel, k sees an empty slot, while j receives a message in that slot. Now two cases are possible.

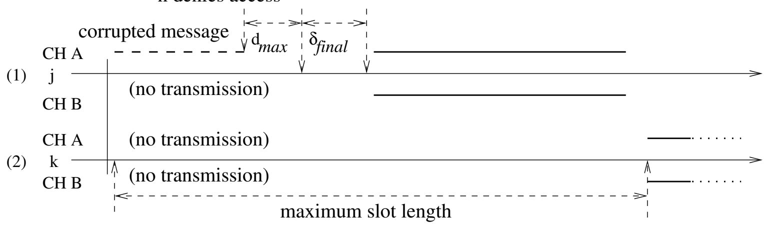


Fig. 6. In case of an asymmetric fault the further activities depend on the next sender. In case (1), j receives a transmission on the faulty channel **A**. If the neighbor of the faulty sender denies access to the bus, j can start immediately. In case (2) k has to wait until the slot ends by reaching the maximum length

1. The neighbor n of f sees an idle transmission. Since the extension method does not allow idle slots, there must be a fault, so n denies further access to the bus. Now any controller sees an idle or aborted transmission. The next controller in the schedule can wait to the maximum slot length, and start sending. This case is also possible if n receives a corrupted message.
2. The neighbor sees a correct message. If the next sender saw this correct message, too, it starts sending. All controllers are receiving now the new transmission, and therefore pass to the next slot. Otherwise the next sender waits until maximum slot length expires.

In any case, the faulty controller cannot disturb the communication. If faults occur, then a maximum slot length may left unused.

3 Examples

In the following two scenarios are investigated. The first one assumes maximum utilization. This means that there are no empty slots in the extension. Note that this is the only possible scenario for agreement-based scheduling in the fault-free case. To preserve comparability, the worst-case scenario consisting of controller and channel faults is excluded. The second scenario shows the overhead in the context of slot usage. The calculations for the assigned slots and agreement-based methods are using the same reference architecture, which is defined by the values $d_{max} = 0.44\,\mu$s and $\delta_{final} = 1.1\,\mu$s (see Figs. 7 and 8). In the following, k is the maximum number of slots in the dynamic parts.

In the minislotting approach with pre-assigned slots, both for the start and the end of a message a full slot must be reserved. Using (1), this gives $T_{as} = k\,2\,\Delta_{max}$.

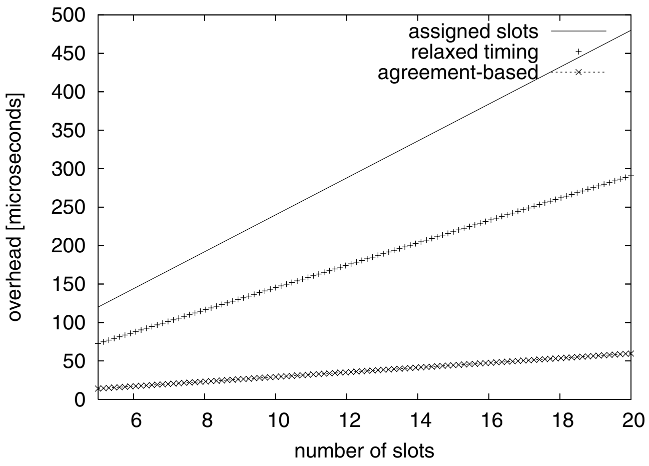


Fig. 7. $\Delta_{max} = 12\,\mu$s, $\epsilon = 1\,\mu$s. These are possible values for a cycle length of 160 ms

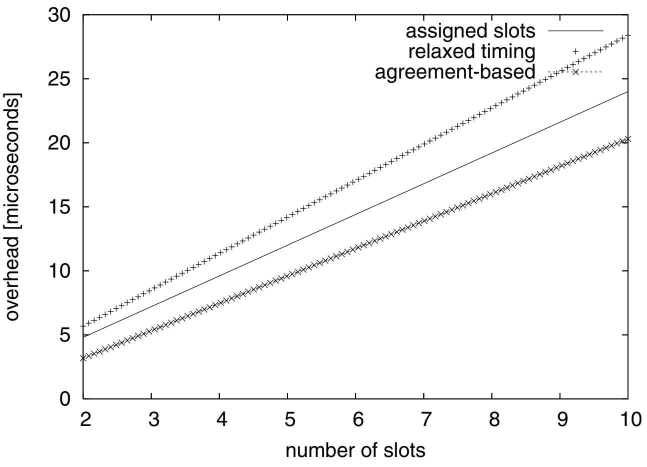


Fig. 8. $\Delta_{max} = 1.2\,\mu$s, $\epsilon = 0.1\,\mu$s. These are possible values for a cycle length of 0.25 ms

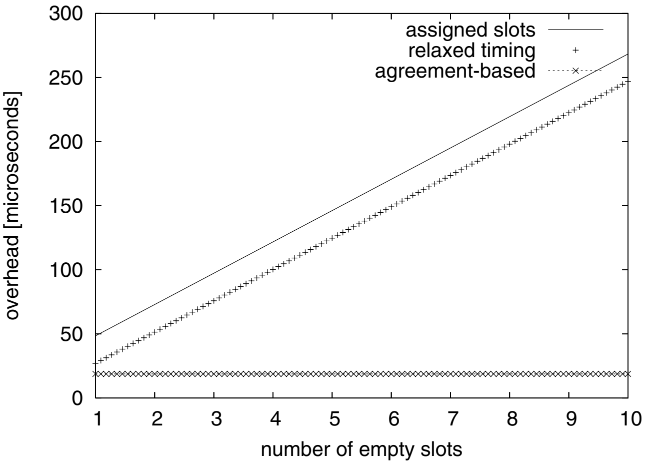


Fig. 9. The overhead in context of a message. Note that the agreement-based method is independent of the number of empty slots

Minislotting with relaxed timing depends on the additional parameter δ_{final} and the distance to the point in the next slot, where the next sender is allowed to send in real time. Therefore the overhead is $T_{ls} = k\,(\delta_{final} + d_{max} + \epsilon + \Delta_{max})$.

With agreement-based scheduling, maximum utilization is granted in the fault-free case, where all senders send messages of maximum length. This means that $h = k$ holds in (3). The total number of controllers should be the same than the number of slots. This allows every controller to get one slot in the extension part. Because the timeout δ_{final} can be left out at the start of the extension part, the overhead is given by $T_{ts} = k\,(d_{max} + d_{switch} + \epsilon + \delta_{final}) - \delta_{final} + T_{req}$.

Figure 7 shows the maximum utilization for different total number of slots. In Fig. 8 the same calculation is shown with a smaller cycle length, which leads to lower values of Δ_{max} and ϵ. The value d_{switch} is set to $0.25\,\mu$s and $d_{bit} = 0.1\,\mu$s.

As one can see, relaxed timing costs lower overhead with long cycles than the assigned slots method. The reason is, that each slot in the assigned slots method must at least as long as twice the maximum offset between the slowest and the fastest clock, which increases with the cycle length depending on the maximum allowed clock deviation. In Fig. 8 this value this value is significantly smaller, so that the impact of the network parameters d_{max} and δ_{final} is dominating. This is possible in the case of a shorter cycle or better clock synchronization. In the agreement-based method the arbitration is based on the agreed schedule and the timeout δ_{final}. Clock synchronization issues doesn't play a role here, because the receivers are synchronized with the messages.

In Fig. 9 the overhead of a sequence of k empty and one non-empty slot is shown. Here, the parameter values of Fig. 7 are used again. For the agreement-based method the worst case is assumed (no assumption of maximum message length). The total number of slots is set to $m = 64$.

The figure shows, that the overhead of the relaxed timing and assigned slots methods are rising with the number of preceding empty slots as expected. The agreement-based method doesn't need to take this into account, because the scheduling is build up before the beginning of the extension part. Since the next sender is already known, it is not necessary to take any arbitration decisions. So there is no reason to introduce additional space.

4 Conclusion

Efficient solutions exist for dynamic arbitration in TDMA-based protocols. While current implementations of strategies with minislotting and pre-assigned slots are able to provide the ability of sending messages of dynamic length, they suffer from their need for bandwidth, because it is never known if the next controller may send or not. Pre-build schedules can bring the need for bandwidth for arbitration purposes to a constant value. Moreover this value could be minimized by enhancing the method for handling messages of dynamic length. However, it is necessary to give away bandwidth in the regular part. In addition, the network remains fully protected during the whole cycle.

References

1. Bauer, G., Frenning, T., Jonsson, A.-K., Kopetz, H., "A Centralized Approach for Avoiding the Babbling-Idiot Failure in the Time-Triggered Architecture", ICDSN 2000, New York, NY, USA
2. Bauer, G., Kopetz, H., Steiner, W., "Byzantine Fault Containment in TTP/C" Proceedings 2002 Intl. Workshop on Real-Time LANs in the Internet Age (RTLIA 2002), pages 13–16

3. M. Peller, J. Berwanger, and R. Griebach. “Byteflight – A New High-Performance Data Bus System for Safety-Related Applications.” BMW AG, EE-211 Development Safety Systems Electronics, 2000
4. The FlexRay Consortium, “FlexRay Communications System – Protocol Specification V2.0” www.flexray.com
5. The FlexRay Consortium, “Bus Guardian Specification V2.0” www.flexray.com
6. Führer T., et al, “Time-triggered Communication on CAN (Time-triggered CAN – TTCAN)” Proceedings of ICC’2000, Amsterdam, The Netherlands, 2000.
7. Heiner, G., Thurner, T., “Time-Triggered Architecture for Safety-Related Distributed Real-Time Systems in Transportation Systems”, In Digest of Papers, The 28th IEEE Int’l Symp. on Fault-Tolerant Computing Systems, Munich, Germany.
8. Kopetz, H., Gruensteidl, G., “TTP – A time-triggered protocol for fault-tolerant real-time systems.” Proceedings 23rd International Symposium on Fault-Tolerant Computing, pages 524–532, 1993.
9. Kopetz, H., “The Time-Triggered Approach to Real-Time System Design” In Predictably Dependable Computing Systems (B. Randell, J.-C. Laprie, H. Kopetz and B. Littlewood, Eds.), Basic Research Series, pp.53-66, Springer, 1995.
10. Kopetz, H., “Real Time Systems – Design Principles for Distributed Embedded Applications”, Kluwer Academic Publishers, 1997
11. Kopetz, H., “A Comparison of TTP/C and FlexRay” Research Report 10/2001 Institut für Technische Informatik, Technische Universität Wien, Austria
12. Kopetz, H., Bauer, G., Poledna, S., “Tolerating Arbitrary Node Failures in the Time-Triggered Architecture” SAE 2001 World Congress, March 2001, Detroit, MI, USA
13. Jens Chr. Lisner, “A Flexible Slotting Scheme for TDMA-Based Protocols” Workshop Proceedings, ARCS 2004, Augsburg, March 26th 2004, pp. 54–65
14. Müller, B., et al, Proceedings 8th International CAN Conference; 2002; Las Vegas, Nv
15. Peller, M., Berwanger, J., Griessbach, R. “The byteflight specification”, V.0.5, 10/29/1999, BMW AG., www.byteflight.com
16. Pease, M., Shostak, R., Lamport L., “Reaching agreement in the presence of faults.” Journal of the ACM, 27(2):228-234, April 1980.
17. Rushby, J., “Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms” Proc. DCCA 6, Garmisch, Germany. IEEE Press. pp. (Preprints) 191-210.
18. Rushby, J., “A Comparison of Bus Architectures for Safety-Critical Embedded Systems” CSL Technical Report, SRI International, 2001
19. Temple, C., “Avoiding the Babbling-Idiot Failure in a Time-Triggered Communication System.” Fault Tolerant Computing Symposium 28, Munich, Germany, June 1998, IEEE Computer Society, pp. 218-227
20. Temple, C., “Enforcing Error Containment in Distributed Time-Triggered Systems: The Bus Guardian Approach” PhD Thesis, Inst. f. Techn. Informatik, Vienna University of Technology.
21. TTTech, “Time-Triggered Protocol TTP/C – High Level Specification Document – Protocol Version 1.1” www.tttech.com, ed.1.4.3, November 19th 2003

An Architectural Framework for Detecting Process Hangs/Crashes

Nithin Nakka, Giacinto Paolo Saggese, Zbigniew Kalbarczyk, and Ravishankar K. Iyer

Center for Reliable and High Performance Computing,
Coordinated Science Laboratory,
University of Illinois at Urbana-Champaign,
1308 West Main St., Urbana IL 61801
{nakka, saggese, kalbar, iyer}@crhc.uiuc.edu

Abstract. This paper addresses the challenges faced in practical implementation of heartbeat-based process/crash and hang detection. We propose an in-processor hardware module to reduce error detection latency and instrumentation overhead. Three hardware techniques integrated into the main pipeline of a superscalar processor are presented. The techniques discussed in this work are: (i) Instruction Count Heartbeat (ICH), which detects process crashes and a class of hangs where the process exists but is not executing any instructions, (ii) Infinite Loop Hang Detector (ILHD), which captures process hangs in infinite execution of legitimate loops, and (iii) Sequential Code Hang Detector (SCHD), which detects process hangs in illegal loops. The proposed design has the following unique features: 1) operating system neutral detection techniques, 2) elimination of any instrumentation for detection of all application crashes and OS hangs, and 3) an automated and light-weight compile-time instrumentation methodology to detect all process hangs (including infinite loops), the detection being performed in the hardware module at runtime. The proposed techniques can support heartbeat protocols to detect operating system/process crashes and hangs in distributed systems. Evaluation of the techniques for hang detection show a low 1.6% performance overhead and 6% memory overhead for the instrumentation. The crash detection technique does not incur any performance overhead and has a latency of a few instructions.

1 Introduction

Usually, a process is said to have *crashed* if it encounters an exceptional condition such that it is no longer able to continue execution. We say a system or process is hung if:

a) it simply does not respond to external inputs, makes no progress, and there is no notification of a failure except as observed by a user or an external monitoring entity, *b)* the process or system goes into an infinite loop[1], detectable only by a dedicated monitor.

The use of heartbeats (HB) to detect failed or hung processes or processors is a common baseline technique [3][18][17], frequently employed in commercial systems. The operating system (OS) can detect most process crashes and detect OS crashes by capturing a wide variety of OS exception conditions and kernel panics, but it is usually unable to detect process hangs and its own hangs. Current approaches detect application/OS hangs by either instrumenting the monitored application to provide an acknowledgment or an "*I am alive*" message, or via a subordinate thread added to the application to enable proactive generation of heartbeats (e.g. Tandem GUARDIAN [26]). The detection of infinite loop conditions is more problematic (without resorting to duplication or self checking mechanisms) since most often the HB thread or the process itself continues to send the "I am alive" message even though the application does not make any progress.

An alternative approach to detect OS crashes/hangs that does not require OS instrumentation uses watchdog timers. A simple user process is employed to periodically reset the watchdog timer, otherwise the OS is considered to be crashed or hung. A limitation of this technique is that an OS crash/hang is indistinguishable from the case where the user-level process that resets the timer abnormally terminates. Another approach used in custom high availability operating systems is to modify the operating system to generate its own heartbeats, again in a separate thread or function [26]. Thus, while it is conceptually simple to assure the existence of a HB mechanism, the design and implementation of a comprehensive HB mechanism is nontrivial. There are three issues which pose significant challenges to HB designers:

Application Intrusiveness. Conventional heartbeat mechanisms detect hangs via augmentation/instrumentation of an application in the form of: (i) *a subordinate thread* – It is possible that the subordinate thread sends heartbeats even if the main process is no longer functional or executes in an infinite loop, or (ii) *progress indicators* (e.g. [21]) – a detailed knowledge of the application and access to the application source code may be required.

Detection Latency. It is essential to have a HB mechanism that detects both process or OS crash/hang with as small a latency as possible. An undetected crash or hang can result in unpredictable behavior of a system and, as a consequence, have a significant negative impact on application or system availability. Since the detection latency depends on a timeout period employed by the HB mechanism, a careful tuning (usually an ad hoc and arbitrary procedure) of the timeout mechanism is required to suit particular application environments and to ensure rapid failure detection.

[1] In our recent study on fault injection based characterization of error sensitivity of Linux kernel on Intel platform (Pentium4 running Linux Kernel 2.4.22), we found that approximately 75% of the manifested errors lead to a crash (easily detectable via heartbeats). Of the remaining 25% of the manifested errors, in about two-thirds of the cases (16%) the processor stopped executing instructions, i.e., processor was frozen. In the remaining one third of the cases (8.3%), the processor continued to execute instructions without doing any useful work e.g., executing in an infinite loop [20].

Determining Timeouts. The problem of determining the timeout values for heartbeats can be addressed using adaptive timeout mechanisms [14][17], although the practical efficiency of these approaches is yet to be determined. Commercial operating systems and distributed or networked environments use an empirical approach to arrive at a fixed timeout value. Timeout values so obtained typically use worst case scenarios, leading to over design and excessive error detection latency. Other solutions used in modern operating systems poll for resource usage [11]; these solutions fail in cases where the process is in an infinite loop without excessively draining any resources. The hardware and performance overhead of self-checking components, as used in the Tandem Guardian system [22] where each of the processes in the process pair heartbeats one another to detect potential hangs, is cost-effective only in high-end server systems (or in switching systems, e.g. AT&T 5ESS Switch [25]).

In overcoming the limitations of the current approaches and addressing the above-mentioned challenges, this paper presents a hardware solution in the form of an in-processor hardware module to support low latency crash and hang detection that has the following unique features:

1. It is operating system neutral.
2. It eliminates the need for any instrumentation to detect all OS Hangs.
3. It provides an automated and lightweight compile-time instrumentation methodology to detect all process hangs (including infinite loops), the detection being performed in the hardware module at runtime.

2 Overview of the Approach

We propose in-processor hardware based techniques to provide low latency crash and hang detection with minimal intrusion to application and low hardware overhead. Three hardware modules are introduced:

The Instruction Count Heartbeat (ICH), which uses existing processor-level features (i.e., performance counters) to monitor whether a processor continues to execute instructions in the context of a specific process. This enables detecting abnormal process termination (i.e., process crashes) and also the detection of process/processor hangs where the process still exists but no more instructions are executed. The ICH does this by monitoring whether a fixed number of instructions of the process are executed within a constant time. It exploits existing performance counters in modern processors and *eliminates any OS and application level instrumentation*. While information regarding the abnormal process termination is available at the OS level, it will be seen that the per-process data collected in the ICH has value in the broader context of locally detecting OS hangs.

The Infinite Loop Hang Detector (ILHD) module detects a process hang due to infinite execution in a legitimate loop. In order to enable this, entry and exit points of loops in the application are instrumented via a compiler (i.e., most compilers are geared to detect loops and can be modified to instrument the application at deterministic points that correspond to the entry and exit points of loops). In addition, an application profiling methodology is employed to determine the timeout values for the heartbeats on a per-loop basis, rather than the usual approach of using a fixed timeout for the entire application.

The Sequential Code Hang Detector (SCHD) module is employed to detect process hang due to infinite execution in illegal loops. An illegal loop can occur when the target address of the terminating branch in the basic block is corrupted and, as a result, the control flow can be subverted to an instruction within the block creating an illegal loop. SCHD detects this scenario by maintaining a log of recently committed instructions and searching for a repetition of an instruction sequence.

The *ILHD* and the *SCHD* require lightweight application instrumentation and achieve high accuracy in determining timeouts. Since the technique automatically instruments the application at loop entry and exit points, it does not require knowledge of the source code.

Fig. 1 depicts: (i) the block diagram of the system with the discussed hardware modules, (ii) the inputs to the modules from the processor pipeline, and (iii) operations performed by the modules on receiving the inputs. Fig. 1 also shows details of an implementation of the ICH Module. The ICH contains multiple sets of counters (*Curr Process Cntr i* and *Prev Process Cntr i*), so as to monitor multiple processes simultaneously (each set of counters being associated with a single process). The *Curr Process Cntr* for a process is incremented when the processor completes execution of a fixed number of instructions (as measured by in-processor performance counters) on behalf of that process. The ICH periodically checks if the *Curr Process Cntr* has been incremented since the previous check. After checking for updates the value in the *Curr Process Cntr* is written to the *Prev Process Cntr*, which is used as a reference for the next check. A separate set of counters, (*Curr OS Counter* and *Prev OS Counter*) are allocated for the OS when it is executing a kernel thread or performing some bookkeeping functions. In addition, a *Counter Array Scan* logic is provided to scan all the process counters and to check for updates on any one of them.

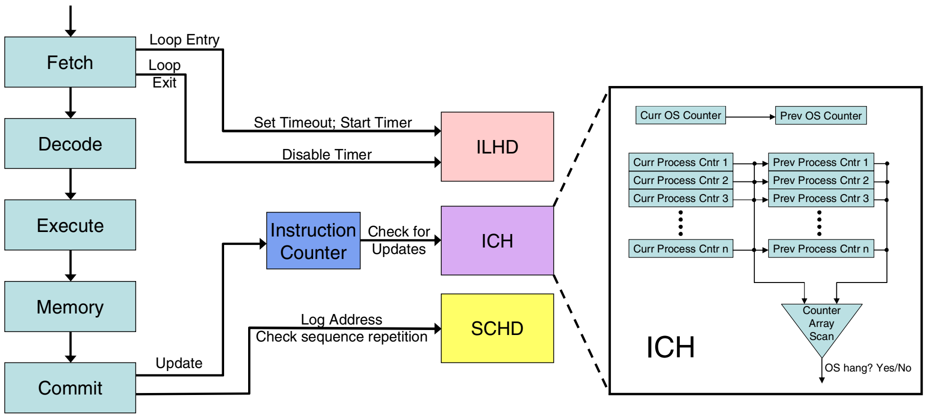


Fig. 1. Block Diagram of Processor with Process Crash/Hang Detection Modules

2.1 Detection of Failure Scenarios

Fig. 2 shows how various failure scenarios can be detected using the techniques introduced in the previous section.

1. Hang when executing user thread
   - *Application Infinite Loop* due to, for instance, design and implementation errors in the application, illegal inputs, or hardware errors that cause the condition for the execution of the loop to be always true – detection is performed by the ILHD or SCHD modules.
   - *System call issued by process never completes* – the detection is performed by the ILHD module, which times out due to the excessive execution time taken by the loop enclosing the system call.

In both cases the module can raise an exception to force the OS to crash/terminate the unresponsive application process.

2. Hang when executing kernel thread.
   - *Processor/Process Frozen* means that the processor/process is not committing any instructions, for instance due to a stuck-at-0 fault on the commit line – the detection is performed by ICH module, which checks the dedicated OS counter to determine lack of progress in executing instructions by the processor.
   - *Kernel Infinite Loop* – the detection can be performed using the *Counter Array Scan* mechanism, which scans all the process counters, at a reasonably long timeout interval, to detect whether the processor stopped executing instructions on behalf of user-level processes. Presence of such conditions indicates that the OS holds on to the processor and the control is never transferred to any user-level process[2].

In the above two cases, resulting in a successful detection, a system reboot is initiated.

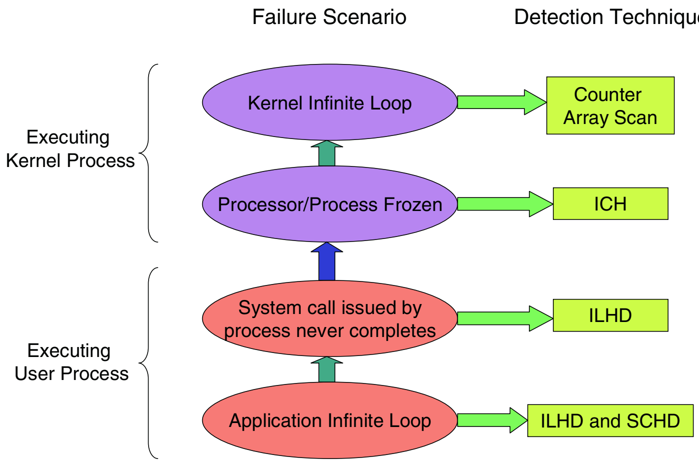


Fig. 2. Hierarchical application of Hang Detection Techniques

[2] If the interrupts are enabled when the OS is in an infinite loop, the detection can also be done by a technique such as the Kernel Health Monitor (KHM) described in [23]. The KHM is a software implemented timer-interrupt driven monitor to detect OS hangs and to initiate a system reboot on successful detection.

At this point one may ask whether the same goals could be achieved by extending the exception handling mechanism of the operating system with corresponding software implementations of the hardware techniques proposed in this paper. While conceptually this might be possible, there are several disadvantages in taking this approach: 1) each change to the exception handling mechanism is OS specific, whereas the hardware approach is OS neutral, and 2) even if the OS mechanism is used to detect process crashes and hangs, a separate support is still needed to detect OS hangs and crashes. The hardware approach is uniquely suited for achieving this goal

Support for Distributed Systems. While the crash/hang detection has value in the context of a single processor, it can also be leveraged to provide efficient failure detection in a clustered or distributed environment. Fig. 3 shows the logic to provide this support (nodes in Fig. 3 correspond to processors). When a distributed application (consisting of a group of processes) initiates a process *p* on a node *A*, it associates a memory mapped I/O port, *P*, (a location in the node *A*'s memory that can be read by a remote process) with the process and initializes the port. The ICH module in node *A* maintains the information of the *process-port* pair. On detecting a crash of a process *p*, the module raises an exception. The OS handles this exception and writes the id of the crashed process to the corresponding port *P*. Remote processes (members of the initial application process group) can read from this port and, hence, detect the crash of process *p*. In case of an application or OS hang, the hang can be detected and transformed into a crash using the procedure described in Section 0.

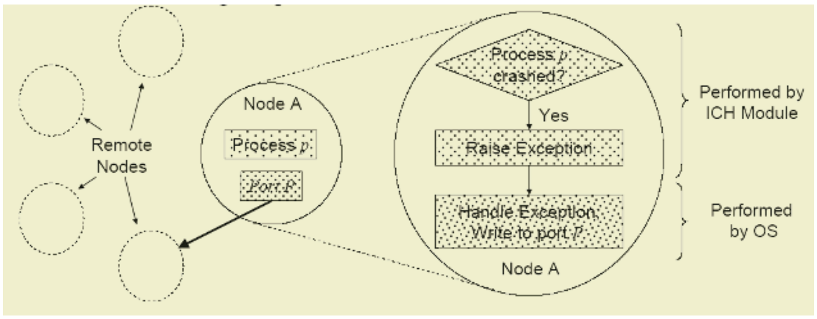


Fig. 3. Crash/Hang detection support for distributed systems

In Summary, the Key Characteristics of the Proposed Approach Are:

- It is hardware implemented and for crash and hang detection requires minimal or no application level instrumentation.
- The crash detection latency can be as small as a few instructions.
- Heartbeats checked concurrently with normal process execution: Since the monitoring entity is a hardware module, the heartbeats from a process *p* are checked concurrently with the execution of *p*. This not only reduces detection latency, but also minimizes the impact of heartbeat on the application performance.

- The scheme provides support in a distributed or networked environment to make failure detectors efficient by *a)* reducing the latency of detection, and *b)* improving the coverage.
- OS neutral: An added advantage of the hardware-based approach over an OS-based one is that it is OS neutral and hence can be leveraged by any OS executing on the hardware.

3 Modifications Needed for the System and Application

All the modules are implemented in hardware and thereby incur hardware overhead. The Infinite Loop Hang Detector (ILHD) and Sequential Code Hang Detector (SCHD) modules require application instrumentation. The aim is to minimize these overheads.

To support the ILHD technique, the application is instrumented with special *CHECK* instructions to notify the monitoring mechanism of the entry and exit points of a loop. Two means of instrumenting the application are explored: (i) use of an *enhanced compiler* – if the application source code is available, a compiler augmented with an additional pass that detects the entry and exit points of loops can be used to embed the CHECK instructions, and (ii) use of a *specialized preprocessor* – if the source code is unavailable, a dedicated preprocessor can be employed to embed the CHECK instructions directly into the application binary (here we leverage our experience using a similar methodology to instrument application binaries with control flow assertion checks [8]). The determination of the timeouts for the loops is done using off-line application profiling. The OS level modifications include support for saving and restoring the state of the modules (timeout values) during a context switch of a process. Fig. 4 shows a flow diagram of the operations to be performed to deploy the ILHD technique.

The implementation of the Instruction Count Heartbeat (ICH) module constitutes a set of timers and counters to wait for heartbeats on completion of execution of a fixed number of instructions. Use of multiple counters enables monitoring multiple processes simultaneously in a multiprogramming environment. Also, since there is no feedback from the module to the processor pipeline, the presence of the module does not affect the instruction execution in the pipeline, hence introducing no performance overhead.

The ICH mechanism requires access to performance counters in the processor that count the number of executed instructions. These performance counters need to be part of the process context state. The application can enable the ICH module from the command line by specifying a predefined option (handled by the OS loader during application invocation). Fig. 5 shows the operations needed at application launch time to apply the ICH technique.

The SCHD module requires instrumentation of the application to parameterize the module for the maximum length of illegitimate loops to be checked. This is explained in greater detail in the following section. The instrumentation can be performed using similar techniques as used for the instrumentation for the ILHD module. The SCHD module requires a set of registers, operating as a queue to log the addresses of the committed instructions. It also requires logic to analyze the log and detect a repetition of an instruction. This module has been described in VHDL, implemented on an FPGA, and detailed hardware overhead analysis performed. As shown in Section 0, the SCHD design shows a hardware overhead of about 5%.

4 Description of Modules for Detection of Process Hangs/Crashes

It is assumed that at the processor level we can distinguish between the processes in the system. Current processors provide registers that allow obtaining a pointer to the process descriptor. For instance, the Intel processor running a Linux Kernel maintains this information in the 13 least significant bits of the `esp` register. This can be used to extract the process id of the current process and, thus, distinguish between the processes in the system.

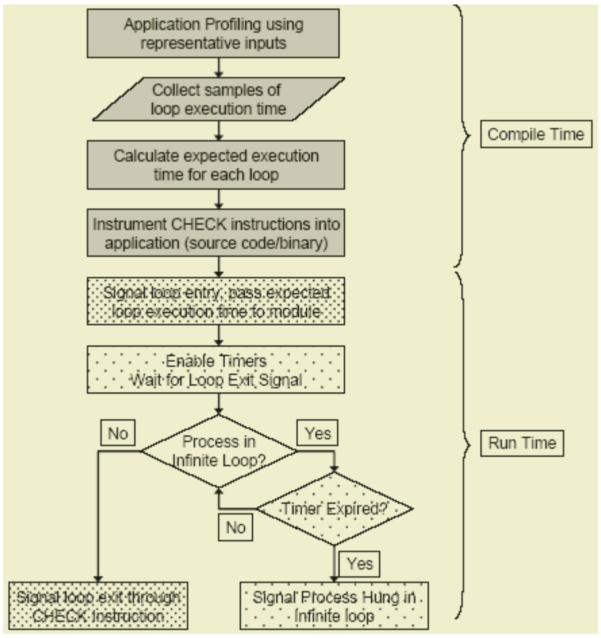

Application Profiling using representative inputs
Collect samples of loop execution time
Calculate expected execution time for each loop
Instrument CHECK instructions into application (source code/binary)
Compile Time
Signal loop entry, pass expected loop execution time to module
Enable Timers Wait for Loop Exit Signal
No
Process in Infinite Loop?
Yes
Timer Expired?
No
Yes
Run Time
Signal loop exit through CHECK Instruction
Signal Process Hung in Infinite loop

Fig. 4. Flow Chart for Deployment of Infinite Loop Hang Detector Module

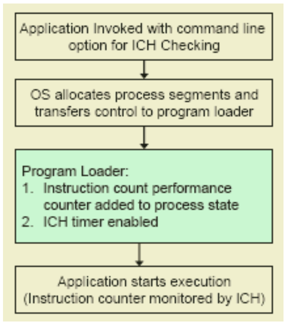
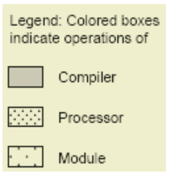

Application Invoked with command line option for ICH Checking
OS allocates process segments and transfers control to program loader
Program Loader:
1. Instruction count performance counter added to process state
2. ICH timer enabled
Application starts execution (Instruction counter monitored by ICH)

Fig. 5. Operations for ICH at process launch



4.1 Detecting Process Crash Using an Instruction Count Heartbeat (ICH)

ICH is a generic technique, applicable throughout the application (irrespective of whether it is executing inside or outside a loop), to detect a process hangs using an estimated time for the application to execute a fixed number of instructions. It does not require any instrumentation of the application. A hardware implementation of the module at the processor level allows monitoring the process for progress at the granularity of number of instructions executed. Thus the detection latency for a process crash would be very low, a multiple of the average estimated instruction execution time.

On modern microprocessors like the Pentium and PowerPC, performance registers can be used to count the number of instructions being committed in the processor. On execution of a certain number of instructions, the processor sends a heartbeat to the ICH module by updating a counter. It checks for updates on this counter, and if the counter is not updated before a timeout occurs, the application is declared to be not executing any instructions or in other words, the application is crashed.

Current generation microprocessors (e.g., Pentium) have a built-in mechanism that monitors the commit sequence of the processor. If the processor has not committed any instructions for a certain time, the pipeline is flushed and restarted. However, this mechanism is not application-specific. Using a separate timer for each process, in which the timer and the instruction counter performance register are part of the process state, makes the technique application-specific. ICH can monitor multiple processes simultaneously and does not require modification of the application itself. The only modification required is to associate the process with a timer and to enable the timer. This can be implemented as a wrapper function executed at invocation, requiring no intrusion into the application.

Although this paper describes the crash/hang detection techniques implemented in hardware, the ICH module can be implemented either as a processor-level hardware module or an OS-level software module. The processor updates the Instruction Counter and the Heartbeat Counter on completion of instruction execution. The ICH module, can be implemented either in the hardware or in the OS layers and check the Heartbeat counter periodically for updates.

Timeout for the Heartbeat. The maximum time required by an application to execute a fixed number of instructions depends on the instruction set architecture of the processor. Events such as cache misses or I/O and networking delays are intercepted, and the timer is disabled for the duration of the event.

The following rules determine the timeout policy:

1. If the heartbeat is received before the timer expires, then the timer is reset and execution continues normally.
2. If the timer expires before the heartbeat is received, the module does not immediately declare the process to be hung/crashed. Instead, the module resets the timer to twice the previous timeout value and waits for the heartbeat.
3. On failure to receive the heartbeat for a threshold (t) number of such exponentially backed-off intervals, the module declares the process to be crashed.

Let T be the average estimated time required to execute an instruction of the processor. Then,

The total time waited before declaring a process crash

$$= \tau = \mathsf{T} + 2 \times \mathsf{T} + 2^2 \times \mathsf{T} + \ldots + 2^t \times \mathsf{T} \ = \mathsf{T} \times (2^{t+1} - 1)$$

A limitation of using a timeout of a fixed number of cycles is that the execution time varies widely from one run to another due to cache misses and different instruction execution latencies. To reduce the effect of cache misses, the timer in the module is disabled while a cache-miss is being serviced. In the other cases, a judicious choice of t and T needs to be made. Next we propose a possible hardware implementation for the ICH module.

Hardware Implementation. For each process being monitored, the module contains two registers, *estTime* (T) and *threshold* (*t*), to store, respectively, the estimated time for the execution of the fixed number of instructions and the threshold for the number of exponentially backed-off intervals before which an application is declared hung. The total time waited τ is calculated using a shift operation on *estTime.* A timer is initialized with this value, and the module waits for the Instruction Counter heartbeat signal from the processor. If the heartbeat is received before the timer expires, then the timer is reset. If the timer expires before the heartbeat is received, the process is assumed to have crashed/hung or entered a deadlocked state.

4.2 Detecting Program Hang Using Infinite Loop Hang Detector (ILHD)

Previous subsections describe hardware-based modules for detecting process crashes that do not require instrumentation of the application. Detection of process hangs on the other hand requires application instrumentation to monitor the application progress. In proposing the Infinite Loop Hang Detector technique we address the two main problems that arise in application instrumentation: (i) placement of instrumented code, (ii) determination of appropriate timeouts. A profiling methodology is employed to monitor the entry and exit points of the loop and derive timeout values on per-loop basis using a profiling methodology. This section describes in detail how the ILHD module achieves these goals.

The problem of finding a program's worst-case execution time is in general undecidable and is equivalent to the halting problem. Significant research work has been done to estimate the loop execution time, since this is a fundamental problem in real time systems. For a program with bounded loops and no recursive or dynamic function calls the loop estimation has been proven to be decidable [9]. Research is in progress for analyzing application with unbounded loops. To detect an application hang, it is assumed that during normal, error-free execution of the program, the execution time for a loop be within a fixed multiple of the execution time in a profile run. If the loop executes for more than this expected time then the application is deemed hung.

Basic Technique. Loop execution time is estimated using static instrumentation to detect loop entry and exit, and dynamic profile information to measure the time elapsed between the arrival of the loop entry and exit signals. During the normal execution of a program, when a loop is entered, a timer is set to the expected execution time of the loop. If the loop is not exited, i.e., the instruction at one of the exit points of the loop is not encountered, before the timer expires, the timer is reset to twice the previous value, waiting for the loop exit condition. If the loop is not exited after a threshold number of such exponentially backed-off intervals, the application is declared to be hung.

Application Instrumentation Through Profiling. In this description, we use terminology for the C programming language to describe loop constructs. The executable dump of the application is statically analyzed to identify all back edges in the program control-flow graph. These are branch or jump instructions with their target address less than or equal to their own address. Each back edge in the program graph is attributed to a loop. There can be more than one exit point for a loop due to a break or a return statement. Each loop is assigned a loop identifier (*loop_id*).

Before each instruction that is the start of a loop or an exit point of the loop, a `CHECK` instruction (`CHECK Loop Start` or `CHECK Loop End`) is inserted to signal the ILHD module of the appropriate event, start of the loop or exit of the loop respectively. Note that a single instruction can be the target for multiple back edges in a program (due to `continue` statements within the loop or due to the loop being executed when one of multiple conditions is true). Similarly, a single instruction may be the exit point for more than one loop.

From the profile run, we obtain samples for the execution times of the loops in the program. The mean (μ_i) and standard deviation (σ_i) for the execution times for loop with *loop_id i* are calculated from this data and stored. Expected execution time for loop *i* is set to be ($\mu_i + n \times \sigma_i$), where *n* is a fixed prespecified number to account for the fact that loop execution times for different inputs can be different.

Runtime Monitoring of Loops in the Application. For the normal run of the program, on encountering the start of a loop, the `CHECK Loop Start` signal is sent to the module. The estimated execution time for the loop is sent through an input register in the module. For loops starting at the same instruction, the execution time for the longest loop (the one with the maximum expected execution time) is sent to the module. One drawback of this approach is that it increases the latency of detecting a hang in the loop. On receiving the `CHECK Loop Start` signal, the module sets a timer with the expected execution time for the loop and waits for the `CHECK Loop End` signal. On failure to receive a `CHECK Loop End` signal after a threshold number of exponential backed-off time intervals, the application is declared hung. To handle a set of nested loops, the technique employs multiple timers.

Fig. 6(a) shows the hardware for detecting an application hang in a loop. The module consists of *d* timers, each associated with a set of two registers, *estTime* and *threshold*, holding the estimated loop execution times and thresholds for the number of exponentially backed-off intervals. Therefore, 2*d* registers are needed. The *shifter.* block is used to calculate the wait time for the loop, which is stored in the register τ_i. The *Load Timer*$_i$ signal is used to initialize the timer, *Timer*$_i$, with the wait time for the loop. *Start Timer*$_i$ is used to start the timer. Fig. 6(b) shows the finite state machine controlling the *d* timers. The technique can only check for loops to a nested depth of *d*. The state of the FSM denotes the nesting depth of the current loop being

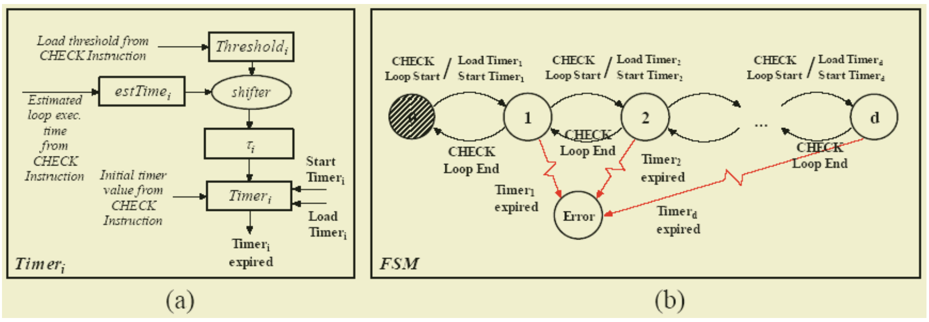

Load threshold from CHECK Instruction
Threshold$_i$
estTime$_i$
shifter
Estimated loop exec. time from CHECK Instruction
τ_i
Initial timer value from CHECK Instruction
Timer$_i$
Start Timer$_i$
Load Timer$_i$
Timer$_i$ expired
Timer$_i$
(a)
CHECK Loop Start / Load Timer$_1$ Start Timer$_1$
CHECK Loop Start / Load Timer$_2$ Start Timer$_2$
CHECK Loop Start / Load Timer$_d$ Start Timer$_d$
0
1
2
d
CHECK Loop End
CHECK Loop End
CHECK Loop End
Timer$_1$ expired
Timer$_2$ expired
Timer$_d$ expired
Error
FSM
(b)

Fig. 6. Infinite Loop Hang Detector Hardware

checked for hangs. It is a Mealy machine, whose transitions are triggered by the `CHECK Loop Start` and `CHECK Loop End` signals. The actions taken by the module on receiving these signals is as explained above

4.3 Detecting Program Hang Using Sequential Code Hang Detector (SCHD)

Apart from causing the application to execute in an infinite loop, an error may also lead to the creation of an illegitimate loop created in sequential code of the application. This may possibly lead to a hang. Repeated execution of a sequence of instructions, while the application is executing sequential code, can be used as a trigger for error detection. The SCHD module detects this scenario minimizing the impact on application performance. This section describes different implementations of the SCHD module of progressively increasing complexity while providing a lower overhead in terms of hardware and performance.

To detect repetition of a sequence of instructions, we maintain a log of the previously committed instructions. If a sequence of instructions at the tail of this log of addresses of committed instructions is repeated, then there is a loop. The coverage of the technique or its ability to detect a program hang depends on the number of previously committed instructions being logged. Currently, it is assumed that the processor can commit at most one instruction per clock cycle. This assumption will be relaxed later in the paper. Consider the sequence of instruction addresses where increasing indices refer to instructions issued later in time:

$$\ldots\ a_k,\ a_{k+1},\ \ldots,\ a_{k+L-1},\ a_{k+L},\ a_{k+L+1},\ \ldots,\ a_{k+2L-1}$$

In this sequence, there is a repetition of a sequence of instructions of length L, starting from position k if and only if:

$$a_{k+i} = a_{k+i+L} \qquad \text{for } i = 0, 1, 2, \ldots, L\text{-}1 \tag{1}$$

A queue is maintained to keep track of the addresses of previously committed instructions. Let W be the width of the address expressed in binary format. Let the number of entries in the queue be D. It is necessary that the depth of the queue D be at least $2L$. Letting k = 0, the contents of the queue are:

$b_0, b_1, \ldots, b_{L-1}, b_L, b_{L+1}, \ldots, b_{2L-1}$ where $b_i = a_{k+i}$ for each i ranging from 0 to $2L$-1

b_{2L-1} is the last entry in the queue, b_{2L-2} is the second last entry and so on.

The condition (1) becomes:

$$b_i = b_{i+L} \qquad \text{for } i = 0, 1, 2, \ldots, L\text{-}1 \tag{2}$$

Consequently, the problem of recognizing a repetition becomes simply one of evaluating condition (2).

Efficient Detection of Sequence Repetition in a Single-Issue Processor. In this section, we detail detection of a sequence repetition that improves upon the previous approach by requiring the minimum queue length to be $L+1$ instead of $2L$. The technique described here detects a repetition of a sequence of length L (or a factor of L). Fig. 7 depicts the hardware used to detect the sequence repetition. It shows a queue that contains the addresses of the instructions most recently committed by the pipeline. The number of entries of the queue is $\geq L+1$. The queue is implemented as a shift register. To insert an element at the tail of the queue, all entries of the queue are shifted up by 1 position: (i+1)st entry = ith entry for i = 1, 2, 3, ... , n-1. The incoming element (address of the committed instruction) is placed in the 1st entry.

A comparator is introduced with its 2 inputs connected to the 1st and the (L+1)st entries in the queue. The output of the comparator is '1' if the inputs are equal, '0' otherwise. The output of the comparator drives the increment and reset inputs of a counter. When a new instruction is added to the queue and the output of the comparator is '1', the counter is incremented (which means the instructions at the 1st and (L+1)st entries match). Otherwise, the counter is reset.

Initially the counter is reset to the value '0'. If a sequence of instructions of length L repeats, the counter would be incremented when each instruction of the repeated sequence is added to the queue and hence the counter would be incremented to a value L. When the counter reaches the value L, the repetition of the sequence is noted and the counter is reset.

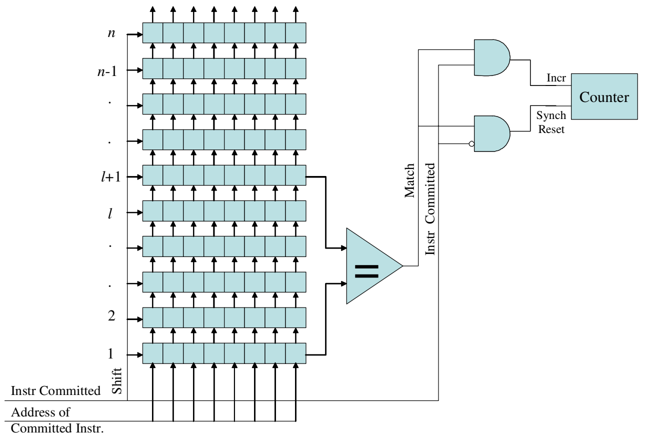


Fig. 7. Basic Sequence Detection Hardware

Detection of Sequence Repetition in a Multiple-Issue Processor. A point to note is that the method described above works only in a single-issue processor. In the case of a superscalar processor that can commit multiple instructions per clock cycle, multiple addresses need to be added to the queue in a single clock, and the pipeline may need to be stalled in order to match the logging of instructions in the module with the committing of instructions in the pipeline. An improved method that can handle multiple committed instructions per clock cycle is presented here. Due to space limitations, we present only the hardware implementation of the technique. Detailed formalization of the technique can be found in [19].

Hardware Architecture of the SCHD Module. We refer to the module detecting a loop of length L as an *L-Loop Searcher* (LLS). Fig. 8 shows the schematic of the *Queue* and of the generic *LLS* module. Let C be the maximum number of instructions that the pipeline can commit in a clock cycle. A SCHD Module detecting any loop whose length is between [L_{min}, L_{max}], requires the following hardware components: (1) a *Queue* module with length (number of entries) $D = L_{max} + C$, (2) (L_{max}–L_{min}+*1*) LLS modules, and (3) an *L'*-input OR-gate, whose inputs are the detect signals, $Detect_L$, coming from the LLS modules. All the modules composing the SCHD Module are clocked by the global clock signal (*clk* in Fig. 8). The enable (*en*), and the reset (*reset*) signals control the memory elements of the design.

The Queue module contains *D* registers, each *W*-bit wide and a Shifter block. The queue can be dynamically configured, through the Shifter block, to enable shifting of *N* positions, where *N* is the number of instructions committed in the current clock cycle by the processor, and ranges from 0 to *C*. The addresses of the committed instructions in a given clock cycle are passed to the queue through the signals in_0, in_1, …, in_{C-1} where the vector *Mask* = ($Mask_{C-1}$, …, $Mask_1$, $Mask_0$) represents which instructions have been committed. The Shifter block is a multiplexer with *C-1* input ports, each with a width of $D{\times}W$ bits, controlled by the *Mask* signal. It shifts the contents of the queue by *N* positions and appends the instructions that have been committed in the current clock cycle. Note that the output of the shifter is forwarded to the LLS modules in order to detect a loop in the same clock cycle as it occurs. Another possibility is that the output of the Queue component can be the output, *u*, of the register file. This breaks the propagation path of the new instructions from the processor to the SCHD module by using the queue as a buffer. This trades off a shorter clock period with a latency of one clock cycle in the detection of a loop. An LLS module is mainly composed of *C W*-wide comparators, a $log_2(L)$-bit wide register, and some other sparse logic.

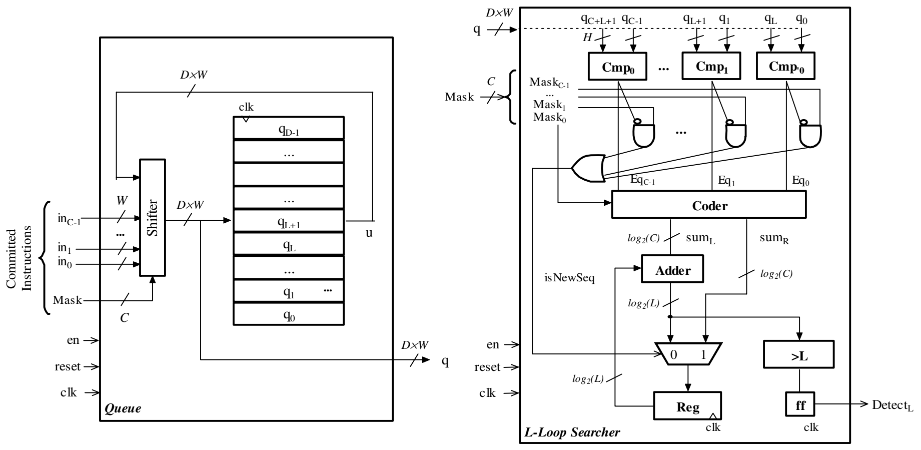


Fig. 8. Hardware Architecture for SCHD Module for Superscalar Processors

5 Implementation and Experimental Results

Two types of experiments have been performed to estimate the overhead in terms of the silicon area for the hardware modules and the overhead for execution time and memory occupation. (1) The ILHD and ICH modules require counters for the timers and registers for holding the execution time parameters. In general, the hardware overhead can be assumed to be negligible, since the modules require simple counters. The SCHD module on the other hand is more complicated, and hence a detailed analysis has been performed. The SCHD module was described in VHDL and synthesized to determine the area overhead. (2) The performance overhead of the ILHD module due to additional `CHECK Loop Start` and `CHECK Loop End` instructions inserted into the instruction stream is calculated in terms of the number of dynamic instructions executed and the execution time.

Area Occupation for the SCHD Module. The area overhead of the SCHD module is evaluated as a percentage of the area required by the processor. In performing this evaluation, we consider a SuperScalar version of the DLX processor [1][7], and implement it using a Xilinx FPGA (VirtexE2000-8) as the target.

The area occupation of the SuperScalar DLX processor is 49% of the overall number of available FPGA slices. The minimum clock period of the synthesized system is about 60 ns. The area occupation of the SCHD is a function of several parameters: (i) the width *W* of the address of the instructions, (ii) the depth *D* of the queue, (iii) the width of the loop range [L_{min}, L_{max}] to search for, and (iv) the number *C* of instructions that the processor can commit in each clock cycle. After the Place-&-Route phase of the design process, we evaluate the area overhead of the Queue, and LLS varying the above mentioned parameters.

We observe that the number of flip-flops required by the Queue module is independent of *C*, while the LUT count increases, since a more complicated Shifter block able to shift a larger range of positions (0 to *C*-1) is needed when *C* increases. Overall, the number of slices required is a small percentage of the overall area available in the device.

The area occupation of the LLS slightly increases as a function of *L*, with *W* and *C* fixed. This is due to the fact that only the Adder, the Comparator, the register and the mux (in the lower part of schematic of the LLS module of Fig. 8) depends logarithmically on *L*.

The area A_{TOT} of a SCHD module is given by the formula:

$$A_{TOT} = A_{QUEUE}(W, D = L_{max} - C, C) + \sum_{i=L\min}^{L\max} A_{LOOP}(W, i, C)$$

where A_{QUEUE}(W, D, C) and A_{LOOP}(W, L, C) is the area required by the queue and the LLS modules respectively. For instance, the area occupation of a SCHD module checking for loops of length ranging between 4 and 32 instructions, for a processor with $C = 2$, and $W = 8$ is about 424 flip-flops, 904 LUTs, and hence an overall area requirement of 526 slices (since we are looking for repetition of a sequence shorter than 32 instructions, the last 8 bits of the address word are sufficient). The area overhead with respect to a SuperScalar DLX processor (which can commit C = 2 instructions) is about 5%. $L_{max} = 32$ is a reasonable number since the module is used to check for loops shorter than the number of instructions in a basic block and is usually 10-20 instructions. The minimum clock period of the module is less than 15ns, which is much smaller than the minimum period of the checked processor, so the time overhead can be considered negligible.

Performance Evaluation. The heartbeat modules are implemented as an augmentation to the *sim-outorder* simulator of the SimpleScalar Tool Suite [2]. The simulator has been modified to support multiple processes executing in a time-sharing manner. In the experiment multiple processes executing the same application are executed in the simulator. The simulator is modified to provide a heartbeat to the ICH module after a process executes a fixed number of instructions. The application is instrumented with `CHECK Loop Start` and `CHECK Loop End` instructions before each entry and exit of a loop to notify the module of the appropriate event. In the profile mode these signals are used to collect execution time data for the loops and calculate an estimate for execution time for each loop is calculated. The overhead due the CHECK instructions in terms of time taken (in number of cycles) is 0.82% and in terms of the number of instructions executed is 2.93%. Noting that we parameterize the SCHD module (for the length of the loop to be detected) using a `CHECK` instruction at the entry and exit of every loop (when it enters straight line code), we conclude that the overhead due to of the SCHD module would be the same as the overhead of the ILHD module amounting to a combined overhead of is 1.6% and 6% with respect to execution time and memory occupation respectively. The ICH module polls the instruction count performance counter in the processor periodically. It does not provide any feedback to the processor and therefore does not affect the instruction execution in the pipeline, incurring no performance overhead.

6 Related Work

Extensive research has been performed in distributed systems to deal with process hang/crash failures (e.g., [11][12][16]). Starting from Chandra and Toeug's seminal paper [3] on characterizing the properties of failure detectors for solving important problems such as Consensus and Atomic Broadcast, failure detectors have been used in distributed systems to detect process crashes in order to circumvent the Fischer-Lynch-Paterson impossibility result [18]. Felber et al. [10] have classified failure detectors into two categories based on their implementation: *push* and *pull.* Though an object framework for interaction between the different entities has been provided, this work does not address the problem of placement of the heartbeat code in the application code of the objects.

Most previous work, specifically commercially implemented heartbeat mechanisms, have used an empirically derived static value of timeout. For example, in the AIX operating system [11], at fixed intervals, a daemon polls the kernel to check if low priority processes are being starved by higher priority processes. In the Microsoft DCOM Architecture [12], clients send periodic "I am alive" messages to servers. If the message is not received within a fixed amount of time, the client is assumed to have crashed and all its resources are de-allocated. In the Sun HA cluster [13] a hierarchy of heartbeat techniques are used to ensure operation of various entities, (servers, nodes, links in private networks, links with public networks). All these mechanisms use empirically derived timeout mechanisms. Heartbeat protocols using adaptive timeouts try to dynamically adapt to the behavior of the application, the system, and the network. Chen [14], Bertier [15], each improving upon the earlier by a slight modification, propose adaptive timeouts that use a linear combination of the previous *n* arrival times as the current timeout. Being implemented in software, these timeouts do not adapt with the changes in the system very well.

A number of heartbeat protocols have been studied, analyzed, and implemented by Gouda and McGuire [4]. The goals of the approach were to decrease the number of heartbeat messages exchanged and the detection latency. Heartbeats have been implemented in both in software and hardware. The heartbeat mechanism is a standard way of detecting node and application failures in most software-implemented middleware, for example the ARMOR Framework [5]. Murphy [6] gives an informal analysis of the use of watchdog timers for monitoring processes. PCI Bus-based watchdog timers can be used to detect system crash [24]. The watchdog timer is reset by a user process that calls a kernel driver. If the timer is not reset for a configurable period of time the system is rebooted. The watchdog timer cannot distinguish between the failure of the user process and a system crash and can only be associated to a single process and cannot detect crashes/hangs of other processes. Table 1 summarizes some of the related work in this area and discusses their limitations.

Table 1. Summary of Related Heartbeat Work

Technique & Description	Questions/Limitations
AIX [11]: Priority Hang Detection Polling to check lowest priority process Detects starvation of low priority processes	Does not detect incorrect control flow (an error in the branch that tests the exit condition of the loop) that leads to process hang in an infinite loop or process crashes. Uses fixed static timeout chosen by the system designer, either empirically or arbitrarily
Microsoft DCOM Architecture [12]: Pinging mechanism to detect Client crash	Cannot detect a client which is hung. Leads to wasted resources that are allocated to the client. Timeout is fixed at 3 × 120 seconds
PVM [16]: PVM daemons notified of task or host failure, host addition.	Does not detect a process hang.
Chen et al.[14]: Estimates the arrival time for the next heartbeat from the previous *n* heartbeat arrival times. Optimizes detection latency and wrong suspicions	The actual triggers to send heartbeat or to reply to a "Are you alive?" ping message are not dealt with. Timeout mechanism is not application specific.
Accrual failure Detector [17]: Output indicates probability that a process has crashed.	Does not interpret the monitoring information. Other limitations same as Chen's technique described above.

7 Conclusions

This paper has explored hardware-implemented techniques for detecting application hang/crash. These techniques are implemented as processor-level hardware modules in order to decrease detection latency. The Instruction Count Heartbeat (ICH) module is a non-intrusive technique for detecting process crashes using instruction count performance counters. For detecting processor hangs, we proposed two techniques that require application instrumentation. The Infinite Loop Hang Detector (ILHD) module detects application hangs within a loop by monitoring the execution time of the application in a loop with respect to an estimated value, derived from profiling. It provides a deterministic methodology for instrumenting the application and determining timeout values. The Sequential Code Hang Detector (SCHD) module detects application hangs due to illegal loops created in sequential code due to errors. The CHECK instruction, a special extension of the instruction set architecture of the processor, is the interface of the application with these modules. In hardware on an FPGA device, we implemented the most complex of the modules, the SCHD module, which showed an area overhead of about 5% with respect to a DLX double-issue superscalar processor. Results for the overhead to support both the ILHD and the SCHD modules simultaneously show low execution time overheads of 1.6% and 6% extra memory occupation.

Acknowledgement

This work was supported in part by Gigascale Systems Research Center (GSRC), NSF grants ACI-0121658 ITR/AP and CNS-0406351 ACI-Next Generation Software Prog., and in part by MURI grant N00014-01-1-0576. We thank T. O' Neill for her careful reading of this manuscript.

References

1. H. Eveking, “SuperScalar DLX Documentation,” http://www.rs.e-technik.tu-darmstadt.de/TUD/res/dlxdocu/DlxPdf.zip.
2. D. Burger and T. M. Austin, “The SimpleScalar Tool Set, Version 2.0,” Tech. Rep. CS-1342, Univ of Wisconsin-Madison, June 1997.
3. T.D. Chandra and S, Toueg, “Unreliable Failure Detectors for Reliable Distributed Systems,” *Journal of the ACM*, New York, v.43, n.2, p.225-267, Mar. 1996.
4. M. Gouda, T. McGuire, “Accelerated heartbeat protocols,” *Proc. of the Int'l Conf. on Distributed Computing Systems,* May 1998, pp. 202-209.
5. Z. Kalbarczyk, S. Bagchi, K. Whisnant, and R. K. Iyer. “Chameleon: A Software Infrastructure for Adaptive Fault Tolerance,” *IEEE Trans. on PDS*, 10(6), June 1999.
6. N. Murphy, "Watchdog Timers," *Embedded Systems Programming*, November 2000.
7. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan-Kaufmann, 1996.
8. Z. Yang, “Implementation of Preemptive Control Flow Checking Via Editing of Program Executables,” Master's Thesis, University of Illinois at Urbana-Champaign, Dec. 2002.
9. Y.-T. S. Li, et al., "Performance Estimation of Embedded Software with Instruction Cache Modeling", *ACM Trans. on Design Automation of Electronic Systems*, 4(3), pp. 257-279.
10. P. Felber, X. Defago, R. Guerraoui, and P. Oser, “Failure Detectors as First Class Objects,” *Proc. of the Int'l Symposium on Distributed Objects and Applications*, 1999.
11. “AIX V 5.1: System Management Concepts,” http://publib16.boulder.ibm.com/pseries/en_US/aixbman/admnconc/syshang_intro.htm.
12. G. Eddon, H. Eddon, “Understanding the DCOM Wire Protocol by Analyzing Network Data Packets,” *Microsoft Systems Journal,* March 1998.
13. “Sun Cluster 3.1 Concepts Guide,” http://docs.sun.com/db/doc/817-0519.
14. W. Chen, S. Toueg, and M. K. Aguilera, “On the Quality of Service of Failure Detectors,” *Proc. DSN* 2000.
15. M. Bertier, O. Marin, and P. Sens, “Implementation and Performance Evaluation of an Adaptable Failure Detector,” *Proc. DSN 2002.*

16. Geist, A.et.al. *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing,* Scientific and Engineering Series. MIT Press, 1994.
17. N. Hayashibaral, X. Defago, R. Yared, and T. Katayama. "The φ Accrual Failure Detector," IS-RR-2004-010, May 10, 2004.
18. M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM),* 32(2):374–382, 1985.
19. N. Nakka, G.P. Saggese, Z. Kalbarczyk, and R.K. Iyer, "An Architectural Framework for Detecting Process Hangs/Crashes," http://www.crhc.uiuc.edu/~nakka/HCDetect.pdf.
20. W. Gu, Z. Kalbarczyk, and R.K. Iyer, "Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors," *Proc. of DSN 2004,* pp. 827-836.
21. K. Whisnant, R.K. Iyer, Z.T. Kalbarczyk, P.H. Jones III, D.A. Rennels and R. Some, "The Effects of an ARMOR-Based SIFT Environment on the Performance and Dependability of User Applications," IEEE Trans. on Software Engg., 30(4), pp. 257-277, April 2004.
22. Inhwan Lee, R. K. Iyer. "Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System," FTCS 1993.
23. D.J. Beauragard. "Error-Injection-Based Failure Profile of the IEEE 1394 Bus," Master's Thesis, University of Illinois at Urbana-Champaign, 2003.
24. "PWDOG1 - PCI Watchdog for Windows XP, 2000, NT, 98, Linux Kernel", http://www.quancom.de/qprod01/homee.htm
25. "AT&T 5ESS™ from top to bottom." http://www.morehouse.org/hin/ess/ess05.htm
26. Daniel P. Siewiorek and Robert S. Swarz. "Reliable Computer Systems: Design and Evaluation", 2nd Edition, Ch. 8.

Energy Efficient Configuration for QoS in Reliable Parallel Servers*

Dakai Zhu[1,**], Rami Melhem[2], and Daniel Mossé[2]

[1] University of Texas at San Antonio, San Antonio, TX, 78249, USA
[2] University of Pittsburgh, Pittsburgh, PA, 15260, USA

Abstract. Redundancy is the traditional technique used to increase system reliability. With modern technology, in addition to being used as temporal redundancy, slack time can also be used by energy management schemes to scale down system processing speed and supply voltage to save energy. In this paper, we consider a system that consists of multiple servers for providing reliable service. Assuming that servers have self-detection mechanisms to detect faults, we first propose an efficient parallel recovery scheme that processes service requests in parallel to increase the number of faults that can be tolerated and thus the system reliability. Then, for a given request arrival rate, we explore the optimal number of active severs needed for minimizing system energy consumption while achieving k-fault tolerance or for maximizing the number of faults to be tolerated with limited energy budget. Analytical results are presented to show the trade-off between the energy savings and the number of faults being tolerated.

1 Introduction

The performance of modern computing systems has increased at the expense of drastically increased power consumption. For large systems that consist of multiple processing units (e.g., complex satellite and surveillance systems, data warehouses or web server farms), the increased power consumption causes heat dissipation problems and requires more expensive packaging and cooling technologies. If the generated heat cannot be properly removed, it will increase the temperature and thus decrease system reliability.

Traditionally, energy management has focused on portable and handheld devices that have limited energy budget to extend their operation time. However, the energy management for servers in data centers, where heat generated and cooling costs are big problems, have caught people's attention recently. In [1], Bohrer presented a case of managing power consumption in web servers. Elnozahy evaluated a few policies that combine dynamic voltage scaling (DVS) [24, 25] on individual server and turning on/off servers for cluster-wide

* Proc. of the Fifth European Dependable Computing Conference, Apr. 2005.
** Work was done while the author was a Ph.D student in University of Pittsburgh.

power management in server farms [5, 14]. Sharma [illegible] investigated adaptive algorithms for voltage scaling in QoS-enabled web servers to minimize energy consumption subject to service delay constraints [19]. Although fault tolerance through redundancy [11, 13, 16, 20] has also been well studied, there is relatively less work addressing the problem of combining fault tolerance and energy management [26, 27]. For systems where both lower levels of energy consumption and higher levels of reliability are important, managing the system reliability and energy consumption together is desired.

Modular redundancy and temporal redundancy have been explored for fault tolerance. Modular redundancy detects and/or masks fault(s) by executing an application on several processing units in parallel and temporal redundancy can be used to re-execute an application to increase system reliability [16]. To efficiently use temporal redundancy, checkpointing techniques have been proposed by inserting checkpoints within an application and rolling back to the last checkpoint when there is a fault [11, 13]. In addition to being used for temporal redundancy, slack time can also be used by DVS techniques to scale down system processing speed and supply voltage to save energy [24, 25]. Therefore, there is an interesting trade-off between system reliability and energy savings.

For independent periodic tasks, using the primary/backup model, Unsal [illegible] proposed an energy-aware software-based fault tolerance scheme which postpones as much as possible the execution of backup tasks to minimize the overlap of primary and backup execution and thus to minimize energy consumption [23]. For Duplex systems, the optimal number of checkpoints, uniformly or non-uniformly distributed, to achieve minimum energy consumption was explored in [15]. Elnozahy [illegible]. proposed an [illegible] (OTMR) scheme to reduce the energy consumption for traditional TMR systems by allowing one processing unit to slow down provided that it can catch up and finish the computation before the deadline if there is a fault [6]. The optimal frequency setting for OTMR is further explored in [28]. Combined with voltage scaling techniques, an adaptive checkpointing scheme was proposed to tolerate a fixed number of transient faults and save energy for serial applications [26]. The work was further extended to periodic real-time tasks in [27].

In this paper, we consider the execution of event-driven applications on parallel servers. Assuming that self-detection mechanisms are deployed in servers to detect faults, for a given system load (i.e., the number of requests in a fixed interval), we explore the optimal number of active servers needed for minimizing system energy consumption while achieving k-fault tolerance. We also explore maximizing the number of faults to be tolerated with limited energy budget. An efficient parallel recovery scheme is proposed, which processes service requests in parallel to increase the number of faults that can be tolerated within the interval considered and thus system [illegible] (defined as the probability of finishing an application correctly within its deadline in the presence of faults [10]).

This paper is organized as follows: the energy model and the application and problem description are presented in Section 2. The recovery schemes are discussed in Section 3. Section 4 presents two schemes to find the optimal number of active servers needed for energy minimization and performability maximization, respectively. The analysis results are presented and discussed in Section 5 and Section 6 concludes the paper.

2 Models and Problem Description

2.1 Power Model

The power in a server is mainly consumed by its processor, memory and the underlying circuits. For CMOS based variable frequency processors, power consumption is dominated by dynamic power dissipation, which is cubicly related to the supply voltage and the processing speed [2]. As for memory, it can be put into different power states with different response times [12]. For servers that employ variable frequency processors [7, 8] and low power memory [17], the power consumption can be adjusted to satisfy different performance requirements. Although dynamic power dominates in most components, the static leakage power increases much faster than dynamic power with technology advancements and thus cannot be ignored [21, 22].

To incorporate all power consuming components in a server and keep the power model simple, we assume that a server has three different states: [illegible], [illegible] and [illegible]. The system is in the [illegible] when it is serving a request. All static power is consumed in the active state. However, a request may be processed at different frequencies and consume different dynamic power. The [illegible] is a power saving state that removes all dynamic power and most of the static power. Servers in sleep state can react quickly (e.g., in a few cycles) to new requests and the time to transit from sleep state to active state is assumed to be negligible. A server is assumed to consume no power in the [illegible].

Considering the almost linear relation between processing frequency and supply voltage [2], voltage scaling techniques reduce the supply voltage for lower frequencies [24, 25]. In what follows, we use [illegible] to stand for changing both processing frequency and supply voltage. Thus, the power consumption of a server at processing frequency f can be modeled as [28]:

$$P(f) = P_s + \hbar(P_{ind} + P_d) = P_s + \hbar(P_{ind} + C_{ef} f^m) \tag{1}$$

where P_s is the sleep power; P_{ind} and P_d are the active powers that are frequency-independent and frequency-dependent, respectively. $\hbar$ equals 1 if a server is active and 0 otherwise. C_{ef} and m are system dependent constants. The maximum frequency-dependent active power corresponds to the maximum processing frequency f_{max} and is given by $P_d^{max} = C_{ef} f_{max}^m$. For convenience, the values of P_s and P_{ind} are assumed to be αP_d^{max} and βP_d^{max}, respectively. Moreover, we assume that continuous frequency is used. For systems that have discrete frequencies, two adjacent frequencies can be used to emulate any frequency as discussed in [9].

Notice that, less frequency-dependent energy is consumed at lower frequencies; however, it takes more time to process a request and thus more sleep and frequency-independent energy is consumed. Therefore, due to the sleep power and frequency-independent active power, there is an energy efficient processing frequency at which the energy consumption to process a request is minimized [28]. Since the overhead of turning on/off a server is large [1], we assume in this paper that the deployed servers are always on and the sleep power P_s is not manageable (i.e., always consumed). Thus, the [illegible] can be easily found as:

$$f_{ee} = \sqrt[m]{\frac{\beta}{m-1}} \cdot f_{max} \tag{2}$$

If $f_{ee} > f_{max}$, that is, $\beta > m - 1$, all requests should be processed at the maximum frequency f_{max} to minimize their energy consumption and no frequency scaling is necessary. Notice that f_{ee} is solely determined by the system's power characteristics and is independent of requests to be processed. Given that f_{low} is the lowest supported processing frequency, we define the minimum energy efficient frequency as $f_{min} = max\{f_{low}, f_{ee}\}$. That is, we may be forced to run at a frequency higher than f_{ee} to meet an application's deadline or to comply with the lowest frequency limitation. However, for energy efficiency, we should never run at a frequency below f_{ee}. For simplicity, we assume that $f_{ee} \geq f_{low}$, that is, $f_{ee} = \kappa f_{max}$, where $\kappa = \frac{f_{ee}}{f_{max}}$.

2.2 Application Model and Problem Description

In general, the system load of an event-driven application is specified by service request[1] arrival rates. That is, the number of requests within a given interval. Although the service time for each individual request may vary, we can employ the law of large numbers and use a mean service time for all requests, which can be justified in the case of high performance servers where the number of requests is large and each individual request has relatively short service time [19]. That is, we assume that requests have the same size and need C cycles to be processed. For the case of large variations in request size, checkpointing techniques can be employed to break requests into smaller sections of the same size [15].

Given that we are considering variable frequency processors, the number of cycles needed to process a request may also depend on the processing frequency [18]. However, with a reasonable size cache, C has been shown to have very small variations with different frequencies [15]. For simplicity, we assume that C is a constant[2] and is the mean number of cycles needed to process a request at the maximum frequency f_{max}. Without loss of generality, the service time needed for each request at f_{max} is assumed to be $c = \frac{C}{f_{max}} = 1$ time unit. Moreover, to ensure responsiveness, we consider time intervals of length equal to D time units. All requests arriving in an interval will be processed during the next interval. That is, the response time for each request is no more than $2D$.

[1] Without causing confusion, we use events and service requests interchangeably.

[2] Notice that, this is a conservative model. With fixed memory cycle time, the number of CPU cycles needed to execute a task actually decreases with reduced frequencies and the execution time will be less than the modeled time.

During the processing of a request, a fault may occur. To simplify the discussion, we limit our analysis to the case where faults are detected through a [illegible] on each server [16]. Since [illegible] and [illegible] faults occur much more frequently than [illegible] faults [3], in this paper, we focus on transient and intermittent faults and assume that such faults can be recovered by re-processing the faulty request.

For a system that consists of M servers, due to energy consideration, suppose that p ($p \leq M$) servers are used to implement a k [illegible], which is defined as a system that can tolerate k faults within any interval D under all circumstances. Let w be the number of requests arriving within an interval D. Recall that the processing of one request needs one time unit. Hence, $n = \lceil \frac{w}{p} \rceil$ time units are needed to process all the requests. Define a [illegible] as the execution of one request on one server. If faults occur during the processing of one request, the request becomes faulty and a [illegible] of one time unit is needed to re-process the faulty request. To tolerate k faults in the worst case, a number of time units, b, have to be reserved as [illegible], where each backup slot has p parallel recovery sections. For a faulty request, the processing during a recovery section may also encounter faults. If all the recovery sections that process a given faulty request fail, then we say that there is a [illegible].

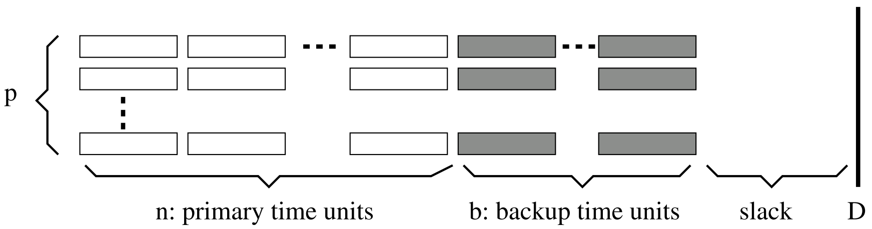


Fig. 1. To achieve a k-fault tolerant system, p servers are used to process w requests within a time interval of D. Here, b time units are reserved as backup slots

The schedule for processing all requests within the interval of D is shown in Figure 1. In the figure, each white rectangle represents a section that is used to process one request on a server and the shadowed rectangles represent the recovery sections reserved for processing the faulty requests. For ease of presentation, the first n time units are referred to as [illegible] and all white rectangles are referred as [illegible]. After scheduling the primary time units and backup slots, the amount of slack left is $D - (n + b)$, which can be used to scale down the processing frequency of servers and save energy.

For a given request arrival rate and a fixed time interval in an event-driven system that consists of M servers, we focus on exploring the optimal number of active servers needed to minimize energy consumption while achieving a k-fault tolerant system or to maximize the number of faults that can be tolerated with limited energy budget.

3 Recovery with Parallel Backup Slots

In this section, we calculate the worst case maximum number of faults that can be tolerated during the processing of w requests by p servers with b backup slots. The addition of one more fault could cause an additional faulty request that can not be recovered and thus leads to a system failure. As a first step, we assume that the number of requests w is a multiple of p (i.e., $w = n \cdot p$, $n \geq 1$). The case of w being not a multiple of p will be discussed in Section 3.4. For different strategies of using backup slots, we consider three recovery schemes: [illegible] and [illegible].

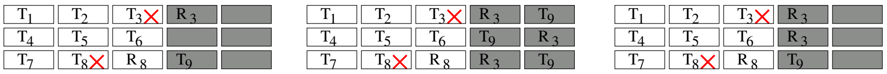


Fig. 2. Different recovery schemes

Consider the example shown in Figure 2 where 9 requests are processed on three servers. The requests are labeled T_1 to T_9 and there are two backup slots (i.e., six recovery sections). Suppose that requests T_3 and T_8 become faulty on the top server during the third time unit and the bottom server during the second time unit, respectively. Request T_8 is recovered immediately during the third time unit (R_8) and the processing of request T_9 is postponed. Therefore, before using backup slots, there are two requests to be processed/re-processed; the original request T_9 and the recovery request R_3.

3.1 Restricted Serial Recovery

The restricted serial recovery scheme limits the re-processing of a faulty request to the [illegible] server. For example, Figure 2a shows that T_3 is recovered by R_3 on the top server while T_8 is recovered by R_8 on the bottom server.

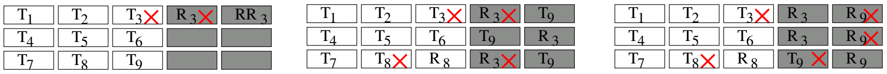


Fig. 3. The maximum number of faults that can be tolerated by different recovery schemes in the worst case

It is easy to see that, with b backup slots, the restricted serial recovery scheme can only recover from b faults in the worst case (either during primary or backup execution). For example, as shown in Figure 3a, if there is a fault that causes request T_3 to be faulty during primary execution, we can only tolerate one more fault in the worst case when the fault causes T_3's recovery, R_3, to be faulty. One additional fault could cause the second recovery RR_3 of request T_3 to be faulty and lead to system failure since the recovery of the faulty requests is restricted to the same server.

3.2 Parallel Recovery

If faulty requests can be re-processed on multiple servers in parallel, we can allocate multiple recovery sections to recover one faulty request concurrently. The parallel recovery scheme considers all recovery sections at the beginning of backup slots and equally allocates them to the remaining requests. For the above example, there are 6 recovery sections in total and each of the remaining requests R_3 and T_9 gets three recovery sections. The schedule is shown in Figure 2b.

Suppose that there are i faults during primary execution and i requests remain to be processed/re-processed at the beginning of the backup slots. With $b{\cdot}p$ recovery sections in total, each remaining request will get at least $\lfloor \frac{b \cdot p}{i} \rfloor$ recovery sections. That is, at most $\lfloor \frac{b \cdot p}{i} \rfloor - 1$ additional faults can be tolerated. Therefore, when there are i faults during primary execution, the number of additional faults during the backup execution that can be tolerated by parallel recovery is:

$$PR(b,p,i) = \left\lfloor \frac{b \cdot p}{i} \right\rfloor - 1 \tag{3}$$

Let $PR_{b,p}$ represents the maximum number of faults that can be tolerated by p servers with b backup slots in the worst case. Hence:

$$PR_{b,p} = \min_{1 \le i \le \min\{b \cdot p, n \cdot p\}} \{i + PR(b,p,i)\} \tag{4}$$

Notice that, w $(= n{\cdot}p)$ is the maximum number of faults that could occur during the n primary time units. That is, $i \le n{\cdot}p$. Furthermore, we have $i \le b{\cdot}p$ because it is not feasible for $b{\cdot}p$ recovery sections to recover more than $b{\cdot}p$ faulty requests. Algebraic manipulations show that the value of $PR_{b,p}$ is obtained when:

$$i = \min\left\{n \cdot p, \left\lfloor \sqrt{b \cdot p} \right\rfloor + u\right\}. \tag{5}$$

where u equals 0 or 1 depending on the floor operation in Equation 3. For the example in Figure 2, we have $PR_{2,3} = 4$ when $i = 2$ (illustrated in Figure 3b) or $i = 3$. That is, for the case shown in Figure 3b, two more faults can be tolerated in the worst case and we can achieve a 4-fault tolerant system. One additional fault could cause the third recovery section for R_3 to be faulty and lead to a system failure. Notice that, although T_9 is processed successfully during the first backup slot, the other two recovery sections in the second backup slot that are allocated to T_9 can not be used by R_3 due to the fixed recovery schedule.

3.3 Adaptive Parallel Recovery

Instead of considering all recovery sections together, we can use [illegible] backup slot [illegible] and adaptively allocate the recovery sections to improve the performance and tolerate more faults. For example, as shown in Figure 2c, we first use the three recovery sections in the first backup slot to process/re-process the remaining two requests. The recovery R_3 is processed on two servers and request T_9 on one server. If the server that processes T_9 happens to encounter a fault, the recovery R_9 can be processed using all recovery sections in the second backup slot on all three servers, thus allowing two additional faults as shown in Figure 3c. Therefore, a 5-fault tolerant system is achieved. Compared to the simple parallel recovery scheme, one more fault could be tolerated.

In general, suppose that there are i requests remaining to be processed/re-processed before using backup slots. Since there are p recovery sections within one backup slot, we can use the first backup slot to process up to p remaining requests. If $i > p$, the remaining requests and any new faulty requests during the first backup slot will be processed on the following $b - 1$ backup slots. If $i \leq p$, requests are processed redundantly using a round-robin scheduler. In other words, $p - i\left\lfloor \frac{p}{i} \right\rfloor$ requests are processed with the redundancy of $\left\lfloor \frac{p}{i} \right\rfloor + 1$ and the other requests are processed with the redundancy of $\left\lfloor \frac{p}{i} \right\rfloor$.

Assuming that z requests need to be processed/re-processed after the first backup slot, then the same recovery algorithm that is used in the first backup slot to process i requests is used in the second backup slot to process z requests; and the process is repeated for all b backup slots.

With the adaptive parallel recovery scheme, suppose that $APR_{b,p}$ is the worst case maximum number of faults that can be tolerated using b backup slots on p servers. We have:

$$APR_{b,p} = \min_{1 \leq i \leq \min\{b \cdot p, n \cdot p\}} \{i + APR(b, p, i)\} \tag{6}$$

where i is the number of faults during the primary execution and $APR(b, p, i)$ is the maximum number of additional faults that can be tolerated during b backup slots in the worst case distribution of the faults.

In Equation 6, $APR_{b,p}$ is calculated by considering different number of faults, i, occurred in the primary execution and estimating the corresponding number of faults allowed in the worst case in backup slots, $APR(b, p, i)$, and then taking the minimum over all values of i. Notice that at most $w = n \cdot p$ faults can occur during the primary execution of w requests and at most $b \cdot p$ faults can be recovered with b backup slots. That is $i \leq \min\{n \cdot p, b \cdot p\}$. Hence, $APR(b, p, i)$ can be found iteratively as shown below:

$$APR(1, p, i) = \left\lfloor \frac{p}{i} \right\rfloor - 1 \tag{7}$$

$$APR(b, p, i) = \min_{x(i) \leq J \leq y(i)} \{J + APR(b - 1, p, z(i, J))\} \tag{8}$$

When $b = 1$ (i.e., $i \leq p$), Equation 7 says that the maximum number of additional faults that can be tolerated in the worst case is $\left\lfloor \frac{p}{i} \right\rfloor - 1$. That is, one more fault could cause a recovery failure that leads to a system failure since at least one request is recovered with redundancy $\left\lfloor \frac{p}{i} \right\rfloor$.

For the case of $b > 1$, in Equation 8, J is the number of faults during the first backup slot and $z(i, J)$ is the number of requests that still need to be processed during the remaining $b - 1$ backup slots. We search all possible values of J and the minimum value of $J + APR(b - 1, p, z(i, J))$ is the worst case maximum number of additional faults that can be tolerated during b backup slots.

The bounds on J, $x(i)$ and $y(i)$, depend on i, the number of requests that need to be processed during b backup slots. When $i > p$, we have enough requests to be processed and the first backup slot is used to process p requests (each on one server). When J $(0 \leq J \leq p)$ faults happen during the first backup slot and the total number of requests that remain to be processed during the remaining $b-1$ backup slots is $z(i,J) = i - p + J$. Since we should have $z(i,J) \leq (b-1)p$, then J should not be larger than $b \cdot p - i$. That is, when $i > p$, we have $x(i) = 0$, $y(i) = \min\{p, b \cdot p - i\}$ and $z(i,J) = i - p + J$.

When $i \leq p$, all requests are processed during the first backup slot with the least redundancy being $\lfloor \frac{p}{i} \rfloor$. To get the maximum number of faults that can be tolerated, at least one recovery failure is needed during the first backup slot such that the remaining $b-1$ backup slots can be utilized. Thus, the lower bound for J, the number of faults during the first backup slot, is $x(i) = \lfloor \frac{p}{i} \rfloor$. Therefore, $\lfloor \frac{p}{i} \rfloor = x(i) \leq J \leq y(i) = p$. When there are J faults during the first backup slot, the maximum number of recovery failures in the worst case is $z(i,J)$, which is also the number of requests that need to be processed during the remaining $b-1$ backup slots. From the adaptive parallel recovery scheme, it is not hard to get $z(i,J) = \left\lfloor \frac{J}{\lfloor p/i \rfloor} \right\rfloor$ when $\lfloor \frac{p}{i} \rfloor \leq J \leq (i - p + i\lfloor \frac{p}{i} \rfloor)\lfloor \frac{p}{i} \rfloor$ and $z(i,J) = (i - p + i\lfloor \frac{p}{i} \rfloor) + \left\lfloor \frac{J-(i-p+i\lfloor \frac{p}{i} \rfloor)\lfloor \frac{p}{i} \rfloor}{\lfloor p/i \rfloor + 1} \right\rfloor$ when $(i - p + i\lfloor \frac{p}{i} \rfloor)\lfloor \frac{p}{i} \rfloor < J \leq p$.

For the example in Figure 2, applying Equations 7 and 8, we get $APR(2,3,1) = 5$. That is, if there is only one fault during the primary execution, it can tolerate up to 5 faults since all 6 recovery sections will be redundant. Similarly, $APR(2,3,2) = 3$ (illustrated in Figure 3c), $APR(2,3,3) = 2$, $APR(2,3,4) = 1$, $APR(2,3,5) = 0$ and $APR(2,3,6) = 0$. Thus, from Equation 6, $APR_{2,3} = \min_{i=1}^{6}\{i + APR(2,3,i)\} = 5$.

3.4 Arbitrary Number of Requests

We have discussed the case where the number of requests, w, in an interval is a multiple of p, the number of working servers. Next, we focus on extending the results to the case where w is [illegible] a multiple of p.

Without loss of generality, suppose that $w = n \cdot p + d$, where $n \geq 1$ and $0 < d < p$. Thus, processing all requests will need $(n+1)$ primary time units. However, the last primary time unit is not fully scheduled with requests. If we consider the last primary time unit as a backup slot, there will be [illegible] d requests that need to be processed after finishing the execution in the first n time units.

Therefore, similar to Equations 3 and 6, the worst case maximum number of faults that can be tolerated with b backup slots can be obtained as:

$$PR_{b+1,p} = \min_{d \leq i \leq \min\{w,(b+1)\cdot p\}} \{i + PR(b+1,p,i)\} \quad (9)$$

$$APR_{b+1,p} = \min_{d \leq i \leq \min\{w,(b+1)\cdot p\}} \{i + APR(b+1,p,i)\} \quad (10)$$

where i is the number of requests to be processed/re-processed on $b+1$ backup slots. $PR(b+1,p,i)$ and $APR(b+1,p,i)$ are defined as in Equations 3 and 8, respectively. That is, we pretend to have $b+1$ backup slots and treat the last d requests that are not scheduled within the first n time units as faulty requests. Therefore, the minimum number of faulty requests to be processed/re-processed is d and the maximum number of faulty requests is $\min\{w, (b+1)\cdot p\}$, which are shown as the range of i in Equations 9 and 10.

3.5 Maximum Number of Tolerated Faults

To illustrate the performance of different recovery schemes, we calculate the worst case maximum number of faults that can be recovered by p servers with b backup slots under different recovery schemes. Recall that, for the restricted serial recovery scheme, the number of faults that can be tolerated in the worst case is the number of available backup slots b and is independent of the number of servers that work in parallel.

Table 1. The worst case maximum number of faults that can be tolerated by p servers with b backup slots

b		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	20
$p=4$	parallel	3	4	6	7	8	8	9	10	11	11	12	12	13	14	**14**	**16**
	adaptive	3	6	10	14	18	22	26	30	34	38	42	46	50	54	58	78
$p=8$	parallel	4	7	8	10	11	12	14	15	16	16	17	18	19	20	20	24
	adaptive	4	10	17	24	31	**39**	**47**	**55**	**63**	**71**	**79**	**87**	**95**	**103**	**111**	**151**

Assuming that the number of requests w is a multiple of p and is more than the number of available recovery sections, Table 1 gives the worst case maximum number of faults that can be tolerated by a given number of servers with different numbers of backup slots under the parallel and adaptive parallel recovery schemes. From the table, we can see that the number of faults that can be tolerated by the parallel recovery scheme may be less than what can be tolerated by the restricted serial recovery scheme. For example, with $p=4$, restricted serial recovery scheme can tolerated 15 and 20 faults when $b=15$ and $b=20$, respectively. However, parallel recovery can only tolerate 14 and 16 faults respectively. The reason comes from the unwise decision of fixing allocation of all recovery slots, especially for larger number of backup slots. When the number of backup slots equals 1, the two parallel recovery schemes have the same behavior and can tolerate the same number of faults.

From Table 1, we can also see that the adaptive parallel recovery scheme is much more efficient than the restricted serial recovery and the simple parallel recovery schemes, especially for higher levels of parallelism and larger number of backup slots. Interestingly, for the adaptive parallel recovery scheme, the number of faults that can be tolerated by p servers increases linearly with the number of backup slots b when b is greater than a certain value that depends on p. For example, with $p=8$, after b is greater than 5, the number of faults that can be tolerated using adaptive parallel recovery scheme increases by 8 when b is incremented. However, for $p=4$, when $b>2$, the number of faults increases by 4 when b is incremented.

4 Optimal Number of Active Servers

In what follows, we consider two optimization problems. First, for a given performability goal (e.g., k-fault tolerance), what is the optimal number of active servers needed to minimize system energy consumption? Second, for a limited energy budget, what is the optimal number of active servers needed to maximize system performability (e.g., in terms of number of faults to be tolerated)? In either case, we assume that the number of available servers is M and that after determining the optimal number of servers p, the remaining $M - p$ servers are turned off to save energy.

4.1 Minimize Energy with Fixed Performability Goal

To achieve a k-fault tolerant system, we may use different number of servers that consume different amount of energy. In the last section we have shown how to compute the maximum number of faults, k, that can be tolerated by p servers with b backup slots in the worst case. Here, we use the same analysis for the inverse problem. That is, finding the [illegible] number of backup slots, b, needed by p servers to tolerate k faults.

For a given recovery scheme, let b be the number of backup slots needed by p servers ($p \leq M$) to guarantee that any k faults can be tolerated. If b is more than the available slack units (i.e., $b > D - \left\lceil \frac{w}{p} \right\rceil$), it is not [illegible] for p servers to tolerate k faults during the processing of all requests within the interval considered. Suppose that $b \leq D - \left\lceil \frac{w}{p} \right\rceil$, the amount of remaining slack time on each server is $slack = D - \left\lceil \frac{w}{p} \right\rceil - b$. Expecting that no faults will occur (i.e., being optimistic), the slack can be used to scale down the primary execution of requests while the recoveries are executed at the maximum frequency f_{max} if needed. Alternatively, expecting that all faults will occur (i.e., being pessimistic), we can use the slack to scale down the primary execution as well as all recovery execution to minimize the expected energy consumption.

Expecting that k_e ($\leq k$) faults will occur (i.e., k_e [illegible]) and assuming that b_e ($\leq b$) is the least number of backup slots needed to tolerate k_e faults, the slack time is used to scale down the primary execution as well as the recovery execution during the first b_e backup slots. The recovery execution during the remaining backup slots is executed at the maximum frequency f_{max} if more than k_e faults occur. Here, optimistic analysis corresponds to $k_e = 0$ and pessimistic analysis corresponds to $k_e = k$. Thus, the k_e [illegible] is:

$$E(k_e) = p \cdot \left[P_s D + (P_{ind} + C_{ef} f^m(k_e)) \frac{\lceil w/p \rceil + b_e}{f(k_e)} \right] \tag{11}$$

where

$$f(k_e) = \min \left\{ \frac{\lceil w/p \rceil + b_e}{D - (b - b_e)}, f_{ee} \right\} \tag{12}$$

is the frequency to process all original requests and the recovery requests during the first b_e backup slots. Recall that f_{ee} is the minimum energy efficient frequency (see Section 2).

Searching through all feasible number of servers, we can get the optimal number of servers to minimize the expected energy consumption while tolerating k faults during the processing of all requests within the interval of D. Notice that, finding the least number of backup slots b to tolerate k faults has a complexity of $\mathbf{O}(k)$ and checking the feasibility of all possible numbers of servers has a complexity of $\mathbf{O}(M)$. Therefore, the complexity of finding the optimal number of servers to minimize the expected energy consumption is $\mathbf{O}(kM)$.

4.2 Maximize Performability with Fixed Energy Budget

When the energy budget is limited, we may not be able to power up all M servers at the maximum frequency. The more servers are employed, the lower the frequency at which the servers can run. Different numbers of active servers will run at different frequencies and thus lead to different maximum number of faults that can be tolerated within the interval considered. In this section, we consider the optimal number of servers to maximize the number of faults that can be tolerated with fixed energy budget.

Notice that, from the power model discussed in Section 2, it is the most energy efficient to scale down all the employed servers uniformly within the interval. With the length of the interval considered being D and with limited energy budget, E_{budget}, the maximum power level that a system can consume is:

$$P_{budget} = \frac{E_{budget}}{D} \tag{13}$$

For active servers, the minimum power level is obtained when every server runs at the minimum energy efficient frequency f_{ee}. Thus, the minimum power level for p servers is:

$$P_{min}(p) = p(P_s + P_{ind} + C_{ef} f_{ee}^m) = p(\alpha + \beta + \kappa^m) P_d^{max} \tag{14}$$

If $P_{min}(p) > P_{budget}$, p servers are not feasible in terms of power consumption. Suppose that $P_{min}(p) \leq P_{budget}$, which means that the servers may run at a higher frequency than f_{ee}. Assuming that the frequency is $f_{budget}(p)$, we have:

$$f_{budget}(p) = \sqrt[m]{\frac{P_{budget}}{p \cdot P_d^{max}} - \alpha - \beta} \tag{15}$$

The total time needed for executing all requests at frequency $f_{budget}(p)$ is:

$$t_{primary} = \frac{\lceil w/p \rceil}{f_{budget}(p)} \tag{16}$$

If $t_{primary} > D$, p servers cannot finish processing all requests within the interval considered under the energy budget. Suppose that $t_{primary} \leq D$. We have $D - t_{primary}$ units of slack time and the number of backup slots that can be scheduled at frequency $f_{budget}(p)$ is:

$$b_{budget}(p) = (D - t_{primary})f_{budget}(p) = D \cdot f_{budget}(p) - \left\lceil \frac{w}{p} \right\rceil \tag{17}$$

From Section 3, the worst case maximum number of faults that can be tolerated by p servers using restricted recovery scheme is $b_{budget}(p)$. For parallel recovery schemes, from Equations 9 and 10, the maximum number of faults that can be tolerated within the interval considered is either $PR_{p,b_{budget}(p)}$ (for the parallel recovery scheme) or $APR_{p,b_{budget}(p)}$ (for the adaptive parallel recovery scheme).

For a given recovery scheme, by searching all feasible numbers of servers, we can get the optimal number of servers that maximizes the worst case maximum number of faults to be tolerated within the interval D.

5 Analytical Results and Discussion

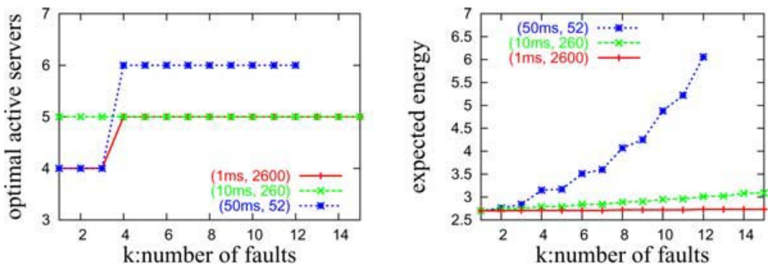
Generally, the exponent m for frequency-dependent power is between 2 and 3 [2]. We use $m = 3$ in our analysis. The maximum frequency is assumed to be $f_{max} = 1$ and the maximum frequency-dependent power is $P_d^{max} = C_{ef} f_{max}^m = 1$. Considering that processor and memory power can be reduced by up to 98% of their active power when hibernating [4, 12], the values of α and β are assumed to be 0.1 and 0.3 respectively. These values are justified by observing that the Intel Pentium M processor consumes $25W$ peak power with sleep power around $1W$ [4] and a RAMBUS memory chip consumes $300mW$ active power with sleep power of $3mW$ [17].

In our analysis, we focus on varying the size of requests, request arrival rate (i.e., system load), the number of faults to be tolerated (k) and the recovery schemes to see how they affect the optimal number of active servers. We consider a system that consists of 6 servers. The interval considered is 1 second (i.e., worst case response time is 2 seconds) and three different request sizes are considered: $1ms$, $10ms$ and $50ms$. The number of expected faults is assumed to be $k_e = \lfloor \frac{k}{2} \rfloor$.

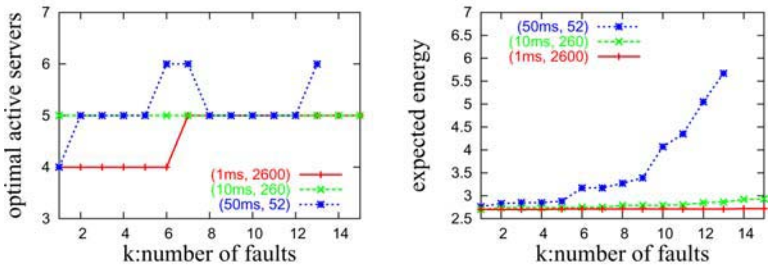
5.1 Optimal Number of Servers for Energy Minimization

Define [illegible] as the ratio of the total number of requests arrived in one interval over the number of requests that can be handled by [illegible] server within one interval. With 6 servers, the maximum system load that can be handled is 6. To get enough slack for illustrating the variation of the optimal number of servers, we consider a system load of 2.6. Recall that the interval considered is 1 second, different request arrival rates are used for different request sizes to obtain the system load of 2.6.

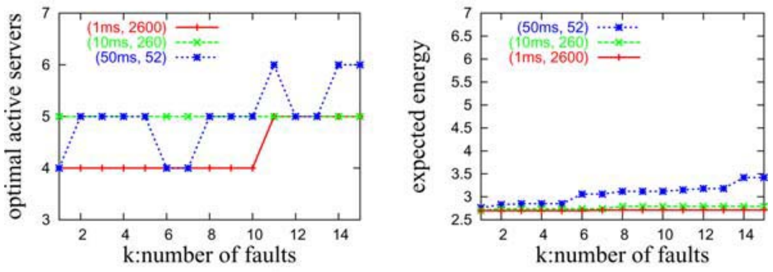
The left figures in Figure 4abc show the optimal number of active servers used (the remaining servers are turned off for energy efficiency) to tolerate a given number of faults, k, under different recovery schemes. The two numbers in the legends stand for request size and request arrival rate (in terms of number of



a. Restricted serial recovery



b. Parallel recovery

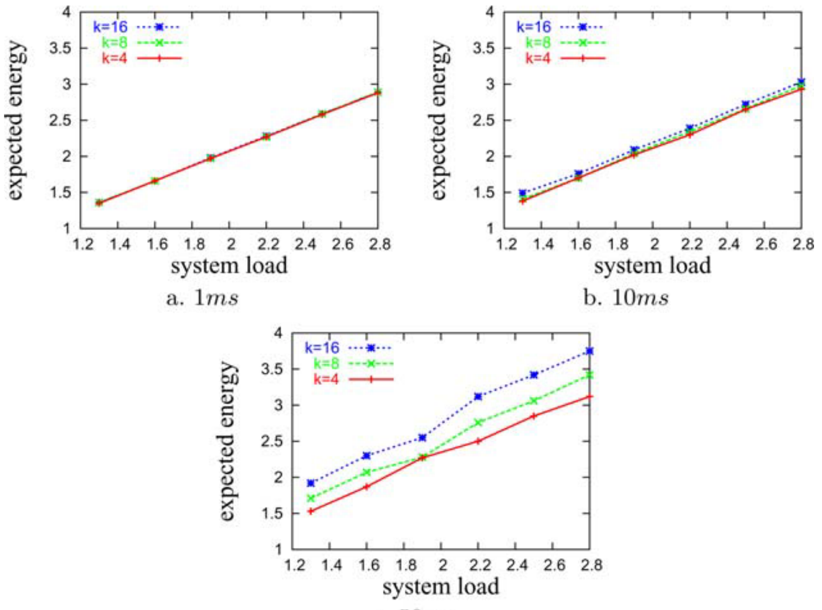


c. Adaptive parallel recovery

Fig. 4. The optimal number of active servers and the corresponding expected minimum energy consumption

requests per second), respectively. From the figure, we can see that the optimal number of servers generally increases with the number of faults to be tolerated. However, due to the effect of sleep power, the optimal number of servers does not increase monotonically when the number of faults to be tolerated increases, especially for the case of large request size where more slack time is needed as temporal redundancy for the same number of backup slots. Moreover, for the case of request size being $50ms$, restricted serial recovery can only tolerate 12 faults and parallel recovery can tolerate 13 faults within the interval considered, while adaptive parallel recovery can tolerate at least 15 faults.

The right figures in Figure 4abc show the corresponding expected energy consumption when the optimal number of servers are employed. Recall that the normalized power is used. For each server, the maximum frequency-dependent



a. $1ms$ b. $10ms$ c. $50ms$

Fig. 5. The minimum expected energy consumption under different system load for different request sizes to tolerate given numbers of faults. The adaptive parallel recovery scheme is used and $k_e = \frac{k}{2}$

power is $P_d^{max} = 1$, sleep power is $P_s = 0.1$ and frequency-independent power is $P_{ind} = 0.3$. From the figure, we can see that, when the request size is $1ms$, the minimum expected energy consumption is almost the same for different numbers of faults to be tolerated. The reason is that, to tolerate up to 15 faults, the amount of slack time used by the backup slots is almost negligible and the amount of slack time used for energy management is more or less the same when each backup slot is only $1ms$. However, when the request size is $50ms$, the size of one backup slot is also $50ms$ and the minimum expected energy consumption increases significantly when the number of faults to be tolerated increases. This comes from the fact that each additional backup slot needs relatively more slack time and less slack is left for energy management when the number of faults to be tolerated increases. Compared with restricted serial recovery and parallel recovery, to tolerate the same number of faults, the adaptive parallel recovery scheme needs fewer backup slots and leaves more slack for energy management. From the figure, we can also see that the adaptive parallel recovery scheme consumes the least amount of energy, especially for larger requests.

For different sizes of requests under adaptive parallel recovery scheme, Figure 5 further shows the expected energy consumption to tolerate given numbers of faults under different system loads. For different request sizes, different request arrival rates are used to obtain a certain system load. When system load increases, more requests need to be processed within one interval and the expected energy consumption to tolerate given numbers (e.g., 4, 8 and 16) of faults increases. As before, when the request size is $1ms$, the expected energy consumption is almost the same to tolerate 4, 8 or 16 faults within the interval of 1 second. The difference in the expected energy consumption increases for larger size of requests.

5.2 Optimal Number of Servers for Performability Maximization

Assume that the maximum power, P_{max}, corresponds to running all servers with the maximum processing frequency f_{max}. When the energy budget for each interval is limited, we can only consume a fraction of P_{max} when processing requests during a given interval. For different energy budgets (i.e., different fraction of P_{max}), Figure 6 shows the worst case maximum number of faults that can be tolerated when the optimal number of active servers are used. The optimal number of active servers increases when energy budget increases but we did not show the results due to space limitation. Here, we consider fixed system load of 2.6. From the figure, we can see that the number of faults that can be tolerated increases with increased energy budget. When the request size increases, there are less available backup slots due to the large slot size and fewer faults can be tolerated. When the number of backup slots is very large (e.g., for the case of $10ms$ with 260 requests/second), the same as shown in Section 3, parallel recovery performs worse than restricted serial recovery. Adaptive parallel recovery performs the best and can tolerate many more faults than the other two recovery schemes at the expense of more complex management of backup slots.

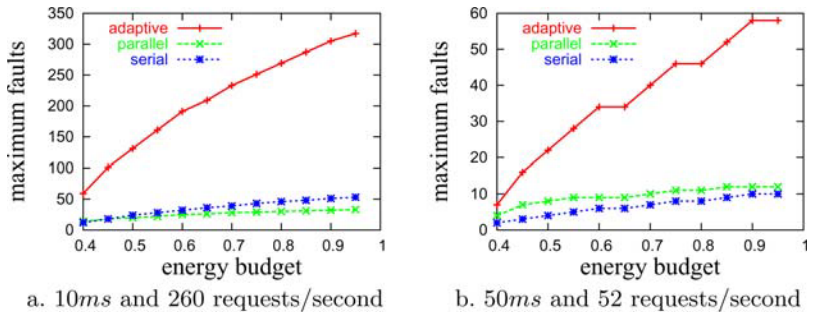


Fig. 6. The worst case maximum number of faults that can be tolerated with limited energy budget for different sizes of requests

6 Conclusions

In this work, we consider an event-driven application and a system that consists of a fixed number of servers. To efficiently use slack time as temporal redundancy for providing reliable service, we first propose an adaptive scheme that recovers requests from faults in parallel. Furthermore, we show that this scheme leads to higher reliability than serial or non-adaptive parallel recovery schemes.

Assuming self-detection mechanisms in each server, we consider two problems that exhibit trade-offs between energy consumption and system performability. The first problem is to determine the optimal number of servers that minimizes the expected energy consumption while guaranteeing k-fault tolerance. The second problem is to maximize the number of faults that can be tolerated with limited energy budget. As expected, our analysis results show that more energy is needed if more faults are to be tolerated. Due to static power consumption in servers, the optimal number of servers needed for k-fault tolerance does not increase monotonically when the number of faults to be tolerated increases. For the same number of faults, large requests will need more slack for recovery and thus is expected to consume more energy. Parallel recovery schemes with a fixed recovery schedule may perform worse than serial recovery. However, adding adaptivity to the parallel recovery process requires less slack to tolerate a given number of faults, leaving more slack for energy management and thus results in less energy being consumed.

When self-detection mechanisms are not available in the system considered, we can further combine modular redundancy and parallel recovery to obtain reliable service. In our future work, we will explore the optimal combination of modular redundancy and parallel recovery to minimize energy consumption for a given performability goal or to maximize performability for a given energy budget.

References

1. P. Bohrer, E. N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony. *The case for power management in web servers*, chapter 1. Power Aware Computing. Plenum/Kluwer Publishers, 2002.
2. T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *Proc. of The HICSS Conference*, Jan. 1995.
3. X. Castillo, S. McConnel, and D. Siewiorek. Derivation and calibration of a transient error reliability model. *IEEE Trans. on computers*, 31(7):658–671, 1982.
4. Intel Corp. Mobile pentium iii processor-m datasheet. Order Number: 298340-002, Oct 2001.
5. E. (Mootaz) Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Proc. of Power Aware Computing Systems*, 2002.
6. E. (Mootaz) Elnozahy, R. Melhem, and D. Mossé. Energy-efficient duplex and tmr real-time systems. In *Proc. of The IEEE Real-Time Systems Symposium*, 2002.
7. http://developer.intel.com/design/intelxscale/.
8. http://www.transmeta.com.
9. T. Ishihara and H. Yauura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. of The 1998 International Symposium on Low Power Electronics and Design*, Aug. 1998.
10. K. M. Kavi, H. Y. Youn, and B. Shirazi. A performability model for soft real-time systems. In *Proc. of the Hawaii International Conference on System Sciences (HICSS)*, Jan. 1994.
11. R. Koo and S. Toueg. Checkpointing and rollback recovery for distributed systems. *IEEE Trans. on Software Engineering*, 13(1):23–31, 1987.

12. A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Proc. of the* 9th *International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.
13. H. Lee, H. Shin, and S. Min. Worst case timing requirement of real-time tasks with time redundancy. In *Proc. of Real-Time Computing Systems and Applications*, 1999.
14. C. Lefurgy, K. Rajamani, Freeman Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, 2003.
15. R. Melhem, D. Mossé, and E. (Mootaz) Elnozahy. The interplay of power management and fault recovery in real-time systems. *IEEE Trans. on Computers*, 53(2):217–231, 2004.
16. D. K. Pradhan. *Fault Tolerance Computing: Theory and Techniques.* Prentice Hall, 1986.
17. Rambus. Rdram. http://www.rambus.com/, 1999.
18. K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. Fast: Frequency-aware static timing analysis. In *Proc. of the IEEE Real-Time System Symposium*, 2003.
19. V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, and Z. Lu. Power-aware qos management in web servers. In *Proc. of the* 24th *IEEE Real-Time System Symposium*, Dec. 2003.
20. K. G. Shin and H. Kim. A time redundancy approach to tmr failures using fault-state likelihoods. *IEEE Trans. on Computers*, 43(10):1151 – 1162, 1994.
21. A. Sinha and A. P. Chandrakasan. Jouletrack - a web based tool for software energy profiling. In *Proc. of Design Automation Conference*, Jun 2001.
22. S. Thompson, P. Packan, and M. Bohr. Mos scaling: Transistor challenges for the 21st century. *Intel Technology Journal*, Q3, 1998.
23. O. S. Unsal, I. Koren, and C. M. Krishna. Towards energy-aware software-based fault tolerance in real-time systems. In *Proc. of The International Symposium on Low Power Electronics Design (ISLPED)*, Aug. 2002.
24. M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Proc. of The First USENIX Symposium on Operating Systems Design and Implementation*, Nov. 1994.
25. F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proc. of The* 36th *Annual Symposium on Foundations of Computer Science*, 1995.
26. Y. Zhang and K. Chakrabarty. Energy-aware adaptive checkpointing in embedded real-time systems. In *Proc. of IEEE/ACM Design, Automation and Test in Europe Conference(DATE)*, 2003.
27. Y. Zhang and K. Chakrabarty. Task feasibility analysis and dynamic voltage scaling in fault-tolerant real-time embedded systems. In *Proc. of IEEE/ACM Design, Automation and Test in Europe Conference(DATE)*, 2004.
28. D. Zhu, R. Melhem, D. Mossé, and E.(Mootaz) Elnozahy. Analysis of an energy efficient optimistic tmr scheme. In *Proc. of the* 10th *International Conference on Parallel and Distributed Systems (ICPADS)*, Jul. 2004.

Novel Generic Middleware Building Blocks for Dependable Modular Avionics Systems

Christophe Honvault[1], Marc Le Roy[1], Pascal Gula[2],
Jean Charles Fabre[3], Gérard Le Lann[4], and Eric Bornschlegl[5]

[1] EADS Astrium SAS, 31 avenue des Cosmonautes,
31402 Toulouse Cedex 4, France
{Christophe.Honvault, Marc.LeRoy}@astrium.eads.net
[2] AXLOG Ingéniérie, 19-21 rue du 8 mai 1945,
94110 Arcueil, France
[3] LAAS-CNRS, 7, avenue du Colonel Roche,
31077 Toulouse, France
[4] INRIA, Domaine de Voluceau - B.P. 105,
78153 Le Chesnay, France
[5] ESA/ESTEC, Keplerlaan 1 - P.O Box 299, 2200 AG Noordwijk,
The Netherlands

Abstract. The A3M project aimed to define basic building blocks of a middleware meeting both dependability and real-time requirements for a wide range of space systems and applications. The developed middleware includes Uniform Consensus (UCS) and Uniform Coordination (UCN) protocols and two services implemented to solve two recurring problems of space applications: “distributed consistent processing under active redundancy” and “distributed replicated data consistency, program serialization and program atomicity”. The protocols have been verified through extensive and accurate testing under the Real Time OS simulator RTSim supporting fault injections. The performances measured on a representative platform based on three LEON SPARC microprocessors interconnected with point-to-point SpaceWire links show that A3M solution may be applied to very different fields, from high performance distributed computing to satellite formation flying coordination.

1 Introduction and Motivations

The complexity of the satellite and launcher on-board data management systems tends to increase rapidly, as well as the strictness of their performance requirements. The efficiency of their development depends on the availability and integration of standard software products (e.g. COTS) solving recurrent problems in space vehicle avionics systems. To address these problems, our objective in this project was to define a basic middleware support for a wide range of space systems and applications. Indeed, the evolution of space platforms is nowadays faced with the inherent distribution of resources and must meet strong real-time and dependability requirements. The handling of combined real-time constraints and fault tolerance strategies raises new problems and calls for new approaches. The A3M (*Advanced Avionics Architecture and Modules*) project was launched to address these issues. The objective was to develop a new generation of space platforms for a wide range of infrastructures, providing generic components as basic building blocks for the development of ad hoc middleware targeting various on-board space applications. The core components of this middleware enable real-time fault tolerant applications to be developed and reused without experiencing the traditional limitations of existing approaches.

The major components of the architecture rely on distributed fault-tolerant consensus and coordination protocols developed at INRIA [1]. These are based on *an asynchronous computational model*, thus making no assumption about underlying timing properties at design time. Therefore, the logical safety and liveness properties always hold without assuming the availability of a global time in the system. The interest of asynchronous solutions for safety-critical systems – choice made by ESA/ESTEC and EADS Astrium – is examined in [2] and explored in details in [3].

These facilities are key features from a dependability viewpoint, as most distributed fault tolerance strategies rely, in one way or another, on this kind of protocols. Their role is essential as far as replicated processing is concerned. For instance, they ensure that all replicas of a given software task receive the same inputs in the same order. Assuming that replicas are deterministic, the processing of the same inputs leads to the same results in the absence of faults. In addition, they are used to ensure that distributed scheduling decisions are consistent in a distributed system as a whole. For instance, in systems where tasks cannot be rolled back (our case), tasks must be granted access to shared persistent resources (updating data in particular) in some total ordering that must be unique system-wide.

In other words, concurrent accesses to shared resources can be serialized by relying on scheduling algorithms based on these protocols. This is a significant benefit, since avoiding uncontrolled conflicts enables real-time deadlines to be met despite failures. One essential assumption regarding failure modes is that nodes in the system fail by crashing, this being a conventional assumption in distributed dependable systems. Note however that the proposed solution may tackle more severe failure modes. Consequently, we assumed a very high coverage of this assumption [4], the means to achieve this being out of the scope of this paper (e.g. evaluation of failure modes as in [5,6]). As usual, it was assumed that no more than f crash failures could be experienced during the execution of a service – middleware-level service in our case.

Application problems are most conveniently tackled at the middleware layer. Such a layer provides appropriate generic services for the development of fault tolerant applications, independently from the underlying runtime support, namely COTS operating system kernel. The core components mentioned earlier constitute the basic layer of this middleware architecture, on top of which standard personalities (e.g. CORBA or better a microCORBA, Java) could be developed. The focus in A3M Phase 2 was the development and the validation (incl. real time characterisation) of this basic layer. The development standard personalities such as CORBA or POSIX are long term objectives not covered by the work reported here.

The paper is organized as follows. Section 2 sketches the A3M architectural framework. In Section 3, we describe the basic principles of the core components, namely UCS and UCN protocols and their variants developed in the project. In Section 4 we describe two key problems in space applications and their corresponding solutions

solutions based on these protocols. Section 5 focuses on the development strategy and justifies its interest from a validation viewpoint. Section 6 addresses the final platform used for the implementation and gives an overview of the A3M results in term of performance. Section 7 draws conclusions of this work.

2 Architectural Framework

The architectural framework of the middleware comprises a basic layer implementing core components for the development of fault tolerant and real-time applications. This basic layer was designed and developed in A3M phase 2 on top of an *off-the-shelf* Real-Time Operating System kernel (RTOS), namely VxWorks [7], and a protocol stack specific to space platforms (COM). The design was such that no particular assumption was made regarding the selected candidates for RTOS or COM. The middleware itself has two facets. On top of the basic layer (facet 1) providing key algorithms, some additional services and personalities can be developed (facet 2) for targeting new problems and new application contexts. The work done until now was to develop the basic layer, i.e. the first facet of the A3M middleware. The overall architecture of the middleware is depicted in Fig. 1.

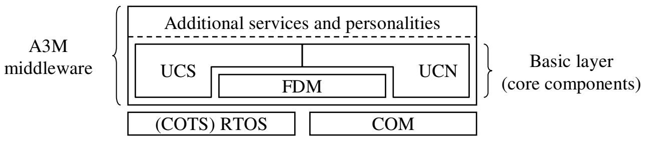

A3M middleware
Additional services and personalities
UCS
UCN
FDM
Basic layer (core components)
(COTS) RTOS
COM

Fig. 1. Architectural framework

The basic A3M middleware layer comprises two major algorithms [1], Uniform Consensus (UCS) and Uniform Coordination (UCN), both relying on a Failure Detection Module (FDM). The A3M middleware layer is also populated with some specific services to tackle problems specific to space applications (see Sect. 4).

3 Basic Components and Protocols

3.1 Uniform ConsensuS (UCS): Basic Principles

UCS is the name of the protocol that solves the Uniform Consensus problem, in the presence of processor crashes and arbitrarily variable delays. UCS comprises two algorithms, one which consists of a single round of message broadcasting, accessible to application programs via a UCS primitive, another called MiniSeq, that runs at a low level (above the physical link protocol level). Both algorithms run in parallel.

UCS works as follows. Upon the occurrence of some event, every processor runs UCS by (1) invoking a best effort broadcasting (BEB) algorithm, which broadcasts a message containing its *Proposal* to every other processor, (2) invoking MiniSeq. A

Proposal may be anything (the name of a processor to be shut down, a list of pending requests to be scheduled system-wide, the name of a processor chosen as the "leader", etc.). Upon termination, UCS delivers a Decision. A Decision must be (1) unique and (2) one of the initial Proposals. Uniformity requires that even those processors that are about to crash cannot Decide differently (than correct processors) before crashing. Note that, in practice, Consensus without the Uniformity property is useless.

The MiniSeq election algorithm, which is sequential, is run by processors according to their relative orderings (based upon their names, from 1 to n, every processor having known predecessors and known successors in set [1, n]). It is via MiniSeq that the Decision is made. MiniSeq uses a Failure Detector, denoted FD. Every processor is equipped with a Strong FD [10]. An FD periodically (every τ) broadcasts FD-messages meaning "I am alive". When processor p has not been heard by processor q "in due time", q puts p on its local list of "suspects". Analytical formulae (see [1]) which express an upper bound (denoted Γ) on FD-message delays are used for setting timers. An interesting feature (F) of Strong FDs is that it suffices to have just one correct processor never suspected by any other processor for solving the Uniform Consensus problem (even though lists of suspects are inconsistent).

Sequentially, every processor waits until some condition fires (namely, its Proposal has been sent out), and then, it broadcasts a Failure Manager message (FM-message), which is an FD-message that contains the name of the processor proposed as the "winner". This name is the name received from the nearest predecessor or its own name if all predecessors are "suspected". The winning value is the value heard last in set [1, n]. The condition for running MiniSeq locally is that every predecessor has been heard of (or is suspected). Thanks to (F), the "winner" is unique system-wide. The Proposal sent by the "winner" is the Decision. Note that we do not assume that broadcasts are reliable (a processor may crash while broadcasting).

The rationale for decoupling BEB and MiniSeq (parallel algorithms) simply is to minimize the execution time of UCS. Indeed, the upper bound B on middleware-level messages that carry Proposals (BEB) is very often much larger than bound Γ on FD-message delays. The execution time of MiniSeq is "masked" by the execution time of BEB whenever some analytical condition is met (see [1]). A simplified condition is as follows: $B > (f+1)\ d$, d given in Sect. 4. There is no a priori ordering of invocations of the UCS primitive (they may be truly concurrent). This above is referred to as *the regular UCS invocation mode*.

Another invocation mode of UCS, called *the Rooted UCS invocation mode* is available. R-UCS serves to make a particular kind of unique Decision, namely whether to abort or to commit a set of updates computed for data variables that may be distributed and/or replicated across a number of processors. R-UCS implements Atomic Commit. The major difference with the regular mode is that instead of having processor number 1 in charge of starting MiniSeq, it is the processor that actually runs the updating task – called the root processor – that is in charge of starting MiniSeq.

3.2 Uniform CoordinatioN (UCN): Basic Principles

The objective UCN is to reach distributed agreement on a collection of data items, each item (a Proposal) being sent by some or all processors. UCN makes use of UCS, starting with a round where every processor broadcasts a message containing a Contribution (to some calculus). At the end of that round, having collected all Contributions sent (up to f Contributions may be missing), every processor runs some computation (called "*filters*" – see below). The result of that computation is a Proposal. UCS is then run to select one unique result that becomes the Decision.

UCN may serve many purposes and can be customized on a case-by-case basis. The algorithm used to establish the final result is called a filter. Such *filter* determines the final result from a collection of input values, namely the proposals. Examples of typical filtering algorithms are: majority voting on input values, aggregation of input values, logical and/or arithmetic expressions on input values, etc. This has many merits since it enables solving several problems such as distributed scheduling.

3.3 Real-Time and Asynchrony

Another innovative feature of the work presented is the use of algorithms designed in the pure asynchronous model of computation augmented with Strong (or Perfect) FDs [10] aimed at "hard" real-time systems. The apparent contradiction between asynchrony and timeliness is in fact unreal, and can be easily circumvented by resorting to the "design immersion" principle (see [2] for a detailed exposition). Very briefly, let us explain the general ideas. With "hard" real-time systems, worst-case schedulability analyses should be conducted. It is customary to do this considering algorithms *designed in some synchronous model*, running in a system S conformant to some synchronous model of computation (S-Mod). It is less customary to do this considering algorithms *designed in some asynchronous model,* running in S, i.e. "immersed" in S-Mod. Within S-Mod, every computation/communication step "inherits" some delay bound proper to S-Mod, this holding true for any kind of algorithm. Hence, the fact that asynchronous algorithms are timer-free does not make worst-case schedulability analyses more complex. One lesson learned has been that the analytical formulae/predictions regarding bounded response times for UCS and UCN were matched with the measurements performed on the A3M platform.

4 Case Studies

A major objective of the A3M study was to assess the interest of UCS/UCN algorithms for space applications. For this purpose, the following two generic and recurrent problems have been selected:

- Problem 1: distributed consistent processing under active redundancy,
- Problem 2: distributed replicated data consistency, program serialization and program atomicity.

This section describes each problem and its solution based on the UCS and UCN algorithms.

Every processor is equipped with a Failure Detection & Management (FDM) module, which consists of an FD module – which broadcasts FD-messages periodically, every τ – and a failure manager (FM), which runs MiniSeq. Locally, at every processor, an FDM module maintains a list of suspected/crashed processors.

Table 1. Hypothesis common to both problems

(F1)	Application level programs are denoted P_k. An instance of every such program is mapped and run, fully, on exactly 1 processor. Programs are assumed to be deterministic.
(F2)	Application level programs, or instances of such programs, can be invoked, activated and run at any time. This means that triggering events can be aperiodic, sporadic or periodic, that programs can be suspended and resumed later, and speed of execution of a program need not be know in advance (asynchronous computational model)
(F3)	There should be no restriction regarding the programming models. In other words, there is no restriction regarding which variable can be accessed by any given program, nor is there any restriction regarding which variables happen to be shared by which program,
(F4)	Neither application level programs nor the system's environment can tolerate the rolling back of a program execution. When a program has begun its execution, it should terminate – in the absence of failure – without replaying some portions of its code.
(F5)	A processor is unambiguously identified by a unique name; names are positive integers; the total ordering of integers induces a total ordering on the set of processors,
(F6)	The system includes n processors. Up to f processors can fail by stopping (crash, no side-effect) while executing some A3M middleware algorithm (UCS, UCN). This number obeys the following condition: $0<f<n$.

"Good" clocks are such that their relative drift is null – or infinitesimal – over "short" time interval, equal to d, the worst-case latency for detecting the occurrence of a processor failure. We have $d = \tau + 2\Gamma - \gamma$, where Γ is a (tight) upper bound on FD-messages transit delays and γ a lower bound on such delays. It is important to note that the processors do not share a common clock (i.e. a common absolute time scale).

In addition, the following assumptions are made about the failures:

- Failures at software level: the software (application software and middleware) is supposed to be "perfect", or monitored by application level failure detectors that halt the processor in case of unrecoverable software error. The middleware is not in charge of application software internal failures detection and recovery. Nevertheless, in the particular case of problem 1, the proposed solution is able to mask a software failure that generates an incorrect result, provided that a majority of processors compute the correct result. Other software failures are not tolerated by the middleware (e.g. common mode failures and Byzantine failures).
- Failures at processor hardware level: the hardware is supposed to have only one failure mode, "crash" having the same effect as a processor halt. Like a halted processor, a crashed processor is no longer able to send any message to the other processors. This implies that the processors have built-in failure detection mechanisms, which is generally the case with space computers.

Failures at communication link or network level: the communication network is supposed to be reliable. This assumption can be "enforced" via classic retransmission-based protocols. For any message the maximum number of retransmissions (denoted

R) needed for delivery is finite and bounded. A message can be lost (and recovered), but not corrupted. With the SpaceWire protocol, this assumption is valid concerning the demonstration platform. If the reliability of the SpaceWire error detection mechanism is not considered as acceptable for a real application, some higher level protocol can be added. If a processor halts or crashes during the transmission of a message, the receiver ignores the partially received message. If the error is only transient, the subsequent messages are transmitted correctly. Of course, analytical formulae for delay bounds such as B or Γ are established taking into account variable R (conventional worst-case schedulability analysis). Recall that we do not assume a reliable broadcast service for neither the BEB nor the MiniSeq algorithms.

4.1 Problem 1: Replicated Processing

The computer system is made of *n* identical computers linked by a network or point-to-point links. The application software can be described as sets of programs, with causal dependencies, i.e. the output of programs are the input of other programs (i.e. a "chain" of application programs – see Fig. 2). External data acquisition comes first in the chain. Two cases are possible for implementing data acquisition (in both cases, the commands are sent by only one processor): only one processor performs the data acquisition or each processor performs the data acquisition in parallel with the others, via its own data acquisition interface.

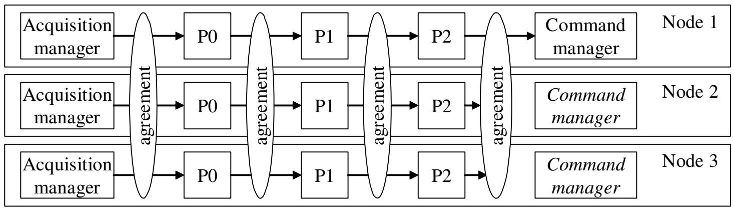


Fig. 2. Replicated processing framework

The same set of programs run in parallel on each computer, exactly in the same way: the output result of each program instance must be validated with respect to those produced by the other replicas, before delivering them as inputs to the next programs in the chain. The middleware allows each Pi replica to make an agreement on input or outputs values by calling dedicated functions. For example, in the case depicted in Fig. 2, P0 calls a function to validate the data produced by the acquisition manager. The measurement may be identical or different on the three processors, but in any case, all the P0 replicas will use as inputs exactly the same values.

The same mechanism is activated after execution of P0, in order to ensure that all the P1 replicas will run with the same inputs, and so on … The use of this agreement service is not limited to a single sequential chain of applications. For example, if P0 and P0' are running "in parallel" replicated on each processor, both can use the agreement service without having to be synchronized.

UCN with a “max filter” solves problem 1. This filter performs a majority selection on contributions. If no majority can be decided, the result is any of the contributions.

Application-level programmers can insert calls for agreement in their code wherever this seems appropriate. The consistency of these calls within application programs must be verified beforehand during the development process of the software.

4.2 Problem 2: Distributed Access to Shared Objects

The second problem consists in replacing the existing “Data-Pool” mechanism used in the current Astrium mono-processor software by an equivalent compatible with the distribution of the processes on several processors – see Fig. 3.

The existing Data-Pool is a collection of data issued from software applications (“software data”) or from equipments connected to avionics buses like MIL1553B (“hardware acquisitions”). In addition to inter process communication, the data pool is used by the periodic housekeeping data reporting applications, by the spy application, and by the monitoring library. Its main purpose is to provide to data consumers consistent data generated from various producers at various frequencies. Concurrent processes running on different processors perform update operations in a pool of data items, called variables. These variables are shared by all processes and replicated for availability reasons. The objective here is to solve the mutual exclusion problem by means of consistent distributed scheduling among competing processes.

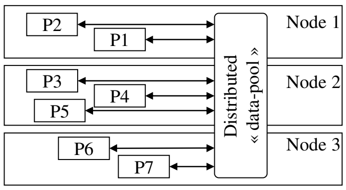


Fig. 3. Distributed access to shared objects

The application tasks use middleware services to ensure the consistency of operations performed on data pool variables sets. The programming model selected for these services is based on the notion of “critical section”. The critical section of an application shall be considered as atomic by all the other applications with respect to the data-pool contents. There is no limitation on the number of read and write accesses performed on data-pool variables in a single critical section. Obviously, the duration of the critical sections should be kept as small as possible, in order to avoid a significant degradation of the overall system performance.
In order to reach this objective, we have introduced two additional services:

– The monitoring service receives from local and remote applications the requests for operations to the distributed data-pool. Using the UCN with a “total ordering filter”, it ensures the system-wide serialization of the conflicting operations on shared variables.

– The guardian service executes the write operation on the replicated data pool, ensuring that any operation is executed on all the copies or on none of the copies. This property is ensured by a variant of UCS, called R-UCS, ensuring atomic program termination (Atomic Commit).

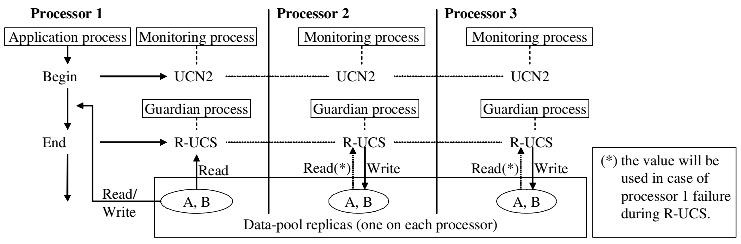


Fig. 4. Process entering a critical section

Fig. 4 illustrates the mechanism in the simple case where only one process wants to enter into a critical section. This process sends a request to the monitoring process that performs an UCN call to agree on which process must run first. Obviously, in this simple case, the requesting process wins, as there are no other competing processes. More generally, thanks to UCN, concurrent requests to enter a critical section are sorted out in the same order on all processors, i.e. all requesting processes will be triggered by the monitoring process in the same order on every processor. The first in the list (according to any urgency criteria) is elected as the next process to be run.

It is worth noting however that, as processes are not replicated here (problem 2), only one processor P will host the active application process, say *p*. When the elected process *p* terminates its critical section on processor P, it signals the guardian process, which then calls R-UCS to perform a consistent update of the replicated copies of shared variables in the data-pool. However, as process *p* was running on processor P only, R-UCS must also be triggered on other processors different from P (those not running a copy of process *p*). This is done via the monitoring process, which directly signals the guardian process, which in turn activates a R-UCS call.

5 Development and Validation

The main originality of the development was to fully develop (in C) the building blocks (BBs) and validate their behavior by simulation, including in the presence of faults. This approach was possible thanks to RTSim, a real-time executive simulator [8]. A distributed framework enabled us to execute complex scenarios, including the creation of an architecture, the setting of a network model, various ways of algorithms stimulation, and temporal and logic fault injection mechanism [9].

5.1 RTSim Overview

RTSim is a product developed by AXLOG. It aims at the development, in host-based environment, of real-time software based on various real-time executives of the market (pSOS+m, VXWORKS, Chorus, ARINC653, etc.), and their running under simulation without any target hardware. By creating nodes, which are the representation of a hardware module running a real-time executive, RTSim could manage simulated architecture from a tiny application running on a single-target system to complex multi-node architecture running a complex distributed application.

One of the main interests of this tool is that it is the real (final) code which is used in the simulation runs, which avoids the problem of abstraction mapping. Moreover, this tool offers advanced debugging, monitoring, controlling and instrumentation facilities over the application. In addition, RTSim has its own internal simulation clock, which enables us to replay a given simulation indefinitely with the same behavior, permitting a fast and reliable way to track defects.

5.2 Development and Simulation Environment

The first phase – the development of the BBs – followed a test-driven development approach (TDD), where each function described by the application programming interface (API) is unitary tested. The main advantages of this approach are:

– a use of the API in tests as in the future user code, giving possible refactoring enhancements before effective coding;
– a test database provides indicators on the current development velocity;
– non-regression tests could be passed easily in case of future implementation.

The second phase was the integrated testing of the BBs (to pass functional validation).

The architecture of the distributed simulation framework used for that purpose, shown in Fig. 5, consists of a multi-node architecture composed of a set of Processor under Test (PuT) representing the target hosting the module to be validated and a special node, the Processor of Test (PoT), responsible for the network simulation, scenarios execution and reporting. All these nodes are connected to a perfect link.

The PoT is in charge of stimulating and controlling the distributed applications, simulating the network, and archiving all actions taken by the system. The PuT hosts the algorithm module, a synthetic applicative workload, and a network service.

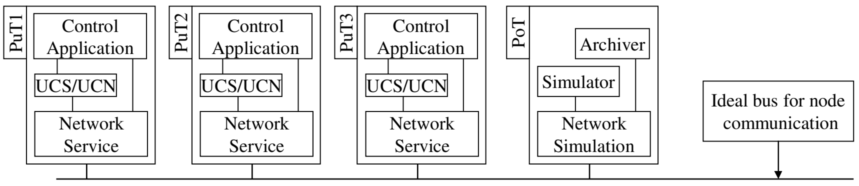


Fig. 5. Simulation framework

5.3 Test Scenarios

After integration of the building blocks in the distributed simulation framework, test scenarios must be identified to cover the entire range of the application use and failure modes. Since UCS and UCN are asynchronous algorithms, time has no effect on their safety and liveliness properties, thus failure occurrence is the only parameter we have to handle (which reduces the complexity of integrated testing very significantly). The algorithms being described by state-machine diagrams, the only concern was to place the failure occurrences in these state diagrams, to derive the scenarios. An analysis of the state diagrams and the effects of a failure have led to simplifying the state diagrams, since the effect of a failure on some contiguous states was unique. The number of test scenarios is a solution to a combinatorial problem, i.e. the assignment of a number of failures varying from 0 to k (k = 2 in our case).

Crash faults have been injected according to this abstract behavioral model, in such a way that all possible states of the protocol are covered. The experiments showed that the implementation was correct according to the considered failure model.

6 Demonstration and Evaluation

The demonstration and evaluation of the middleware has been performed on a platform representative of the next generation of on-board computer platform.

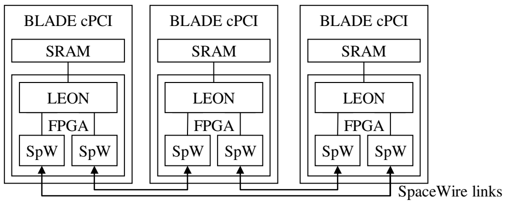


Fig. 6. The A3M Platform

This platform (fully operational at ESA/ESTEC) mainly consists in three double Europe Blade boards plugged in a CompactPCI cabinet. Each Blade board includes a FPGA programmed with a LEON processor and is connected to the two others by a SpaceWire link, and its two serial ports are connected to a RS232/Ethernet adapter. The SpaceWire communication software has been tested, and its performance measured. The raw performance of the hardware is pretty good (30 Mbits/s from memory to memory with a bit rate equal to 40 Mbit/s, which correspond to 99.5% of practical data rate), but the software mechanisms required at communication level by the middleware algorithms decrease the data transfer rate.

The test software consists in several tasks that can run either fully replicated on, or distributed across, the three processors. The objectives of the tests were to perform accurate performance measurements and check the behavior of the system in nominal conditions *and* in the presence of failures. To match these objectives, the duration of the different services of the middleware and operating system was measured with a great precision, i.e. less than 4µs. Computation errors and failures could be generated at predefined times.

To obtain a trustworthy characterization of the middleware, the tests have been executed on different platform configurations, with diverse parameters for FD algorithms (τ, γ and Γ) to be representative of existing spaceborne networks and with different size of exchanged data. The analysis of the measurements also had to take into account the resolution of the operating system timer, which has an negative impact on the worst-case latency for detecting the occurrence of a processor failure. These parameters are summarized in Table 2.

Table 2. Test configuration parameters (time expressed in milliseconds and size in bytes)

Test parameters	Set 1	Set 2	Set 3	Set 4
Processor number	1 to 3			
Variable size	40			272
Variable set size (2 variables)	80			544
Maximum number of tasks	16			
τ (FD messages period)	1000	100	50	1000
γ (min. transmission time of the FD message):	100	0	0	100
Γ (max. transmission time of the FD message)	1000	18	18	1000
ε (operating system timer resolution)	17			

The measures performed during the tests are compared with the theoretical maximum delay *d* for the detection of a failure on a processor given by the formula:

$$d = \tau + 2.\Gamma - \gamma + 2.\varepsilon\,. \qquad (1)$$

Test of Service 1: Distributed Consistent Processing Under Active Redundancy. This service is supported by a single function belonging to the middleware API, "*int dp1_check(dp_varset_id varset_id)*". Several processes running on the processors call the service to make a consensus on a set of values to use at the next step of a computation. The duration of the computation may vary from a processor to another to simulate a diversified programming.

Table 3. Measures performed during test executions of service 1

Nb of Proc.	Param. set	Service duration (ms)		
		Without failure		At failure
		Best	Worst	Worst
3	1	7.5	14.5	1300
2	1	5.5	7.8	1200
1	1	0.5	0.6	N/A
3	2	8	12	222
2	2	5.9	6.8	528
1	2	1.4	1.5	N/A
3	2	8	10	-
3	3	7.6	12.8	82.5
2	3	5.7	9.2	92.5
3	4	11	16	-

The results of the tests show the correct behaviour and performances of the middleware that are in accordance with the underlying theory. The WCET for the failure detection delay is never reached and errors are corrected without overhead. Moreover, the size of the exchanged data is not the main factor of service duration. By exchanging nearly 700% more data, the service duration only increases by 35%.

Test of Service 2: Distributed Replicated Data Consistency. This service is supported by a couple of functions belonging to the middleware API, "*int dp2_begin(dp_task_id task_id, dp_varset_id varset_id)*" and "*int dp2_end(dp_task_id task_id)*" respectively used to enter and leave a critical section associated to a set of variables. Several processes running on the processors call the service to access shared variables. The duration of the reservation may vary. The next table shows some of the measures performed during test executions.

Table 4. Measures performed during test executions of service 2

Nb of Proc.	Param. Set	Operation	Service duration (ms)		
			Without failure		At failure
			Best	Worst	Worst
3	1	begin	10,5	12	1805
		end	6	7,2	1929
3	2	begin	10,5	11	130,5
		end	6,2	10	240
2	1	begin	9,5	10,5	1921
		end	5,8	6,8	1628
2	2	begin	9,25	11,1	786,5
		end	6,3	7,5	404
3	3	begin	9,5	11,5	93
		end	6,3	7,6	59,8
3	4	begin	10,6	11,8	786,5
		end	7,6	10	404

In all the tests, the service fulfils its specifications. The data consistency is ensured on all processors and the priority for accessing them is respected. The WCET for the failure detection delay is never reached. As for service 1, the duration of the procedures is weakly tight to the size of the exchanged variables.

In additional tests, not presented here, the two middleware services have run concurrently and the results obtained were totally consistent.

7 Conclusion and Perspectives

The aim of the A3M project is to investigate and develop a new generation middleware for space applications. The distributed nature of the space platforms, the real-time requirements of the on-board applications together with stringent dependability constraints call for novel architectural frameworks and core services. The full picture also includes development processes and tools at various stages of the design and the implementation of a system. As far as real-time constraints are concerned, a detailed specification of the application organization, the shared resources (variables, critical sections, etc.) and their timing behavior must be built first. Then, a detailed analysis aimed at establishing timeliness properties must be conducted a priori. This entails analyzing middleware-level algorithms, such as schedulers, and how they cope with concurrency, failures and timing constraints.

The work carried out in A3M Phase 2 led to the development of the basic sub-layer of the middleware. The core components rely on UCS and UCN-based modules devoted to solving conventional problems in space applications, ensuring fault-tolerance and real-time behavior in a distributed environment.

This work is the starting point for the development of middleware-based platforms for space applications. It demonstrates the benefits of advanced research results in this context and calls for further development of middleware-based systems for dependable modular avionics systems. It also shows the significant interest of simulation-based approaches for the development of basic middleware services and their validation, including in the presence of faults using fault injection techniques. This work should be continued, possibly with other industrial partners and targeting different application contexts. The results obtained are very promising and the development of additional middleware capabilities should be very attractive for the space industry at large.

References

1. J.-F. Hermant, G. Le Lann, Fast Asynchronous Uniform Consensus in Real-Time Distributed Systems », *IEEE Transactions on Computer*s, vol. 51(8), August 2002, pp. 931-944
2. G. Le Lann, Asynchrony and Real-Time Dependable Computing, Proceedings of the 8th *IEEE Intl. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS*), January 2003, Guadalajara, Mexico
3. G. Le Lann, U. Schmid, How to Maximize Computing Systems Coverage, *Technical Report 183/1-128,* Department of Automation, Technical University, Vienna (Austria), April 2003
4. D. Powell, Failure mode assumptions and assumption coverage, *22nd IEEE Annual International Symposium on Fault-Tolerant Computing (FTCS-22)*, Boston (USA), 8-10 Juillet 1992, pp.386-395. [Revised version in Predictably Dependable Computing Systems, Springer, ISBN 3-540-59334-9, 1995, pp.123-140]
5. J.-C. Fabre, F. Salles, M. Rodríguez Moreno and J. Arlat, Assessment of COTS Microkernels by Fault Injection, *in Proc. 7th IFIP Conf. on Dependable Computing for Critical Applications (DCCA-7),* San Jose, January 1999, pp. 25-44 .
6. J. Arlat, J.C.Fabre, M. Rodriguez, F. Salles: Dependability of COTS Microkernel-based Systems, *IEEE Transactions on Computers*, Special Issue on Embedded Fault Tolerant Systems, Feb. 2002, pp.138-163.
7. VxWorks Realtime Kernel, WindRiver Systems, 1998. (see http://www.windriver.com/products/platforms/general_purpose/)
8. RTSIM real-time executives simulator, AxLog. (see http://www.axlog.fr/prod/rtsim.html)
9. M.-C. Hsueh, T. K. Tsai and R. K. Iyer: Fault Injection Techniques and Tools, *Computer*, vol. 30, no. 4, pp. 75-82, April 1997.
10. T. Chandra, S. Toueg, " Unreliable Failure Detectors for Reliable Distributed Systems", *Journal of the Association for Computing Machinery*, Vol. 43, N. 2, Mar. 1996, pp 225-267.

Integrating Fault Tolerance and Load Balancing in Distributed Systems Based on CORBA*

A.V. Singh, L.E. Moser, and P.M. Melliar-Smith

Department of Computer Science,
Department of Electrical and Computer Engineering,
University of California, Santa Barbara, CA 93106
avsingh@cs.ucsb.edu
{moser, pmms}@ece.ucsb.edu

Abstract. Fault tolerance and load balancing middleware can increase the quality of service seen by the users of distributed systems. Fault tolerance makes the applications more robust, available and reliable, while load balancing provides better scalability, response time and throughput. This paper describes a software infrastructure that integrates fault tolerance and load balancing within a distributed system based on CORBA. The software infrastructure employs Eternal's FTORB, which replicates CORBA applications and thus makes them fault tolerant, and TAO's Load Balancer, which balances the load of the clients' connections across multiple instances of a CORBA server.

1 Introduction

As distributed applications are deployed more widely, the need for improved scalability, response time and throughput becomes more important. An effective way to address this need is to employ a load balancer, based on distributed object middleware, such as the Common Object Request Broker Architecture (CORBA). Also important is the availability, reliability and robustness of the services that the applications provide and, thus, fault tolerance is essential. Fault tolerance employs replication to mask faults and provide continuous service to the users.

Fault tolerance and load balancing can be thought of as orthogonal aspects of quality of service. Both fault tolerance and load balancing require the availability of multiple computers, which distributed systems provide. Many industries with mission-critical applications, such as telecommunications, financial, aerospace and defense, need both fault tolerance and load balancing within a single integrated infrastructure.

The Object Management Group has developed specifications for fault tolerance [7] and load balancing [8] based on CORBA. In this paper, we describe a software infrastructure that integrates fault tolerance and load balancing for CORBA-based distributed systems. We discuss challenges that we had to address in the integration, and present performance results that we obtained for the integrated infrastructure.

* This research has been supported by DARPA/ONR Contract N66001-00-1-8931 and MURI/AFOSR Contract F49620-00-1-0330.

2 The Fault Tolerance and Load Balancing Infrastructure

Our fault tolerance and load balancing infrastructure is based on Eternal Systems' FTORB and TAO's Load Balancer, both of which support the industry-standard CORBA distributed object computing standard. First, we describe each of those middleware components and, then, we describe our fault tolerance and load balancing infrastructure, which integrates those components.

2.1 Eternal's FTORB

Eternal's FTORB [6] provides fault tolerance by replicating servers, clients and infrastructure components (*e.g.*, the Load Balancer) that are implemented as CORBA objects, using active or semi-active replication. FTORB replicates the state of an object in volatile memory, rather than on stable storage, to eliminate the time required to write (read) that state to (from) disk. Research on integrating replication with transactions and databases, where state is persisted to disk, can be found in [13].

FTORB is not itself an ORB but is middleware that works with CORBA ORBs that support the Internet Inter-ORB Protocol (IIOP) running over TCP/IP, with no modification to the ORB. IIOP allows clients and servers that run over different ORBs to communicate with each other. Fig. 1 shows the components of Eternal's FTORB middleware. The functionality of each component is described below:

- **Interceptor:** The Interceptor Library, coupled with each client or server in a fault tolerance domain, intercepts messages and diverts them to the Eternal Replication Engine and the Totem Multicast Protocol, instead of sending them over TCP/IP.
- **Replication Engine:** The Replication Engine maintains groups of client and server replicas, and interacts with the Interceptor Library.
- **Totem Multicast Protocol:** The Totem Multicast Protocol [5] interacts with the Replication Engine and multicasts the clients' requests and servers' replies to the client and server groups, using a logical token-passing ring. Totem can be replaced with any other reliable totally-ordered multicast protocol, such as the Rose rotating sequencer protocol which is also deployed with FTORB.

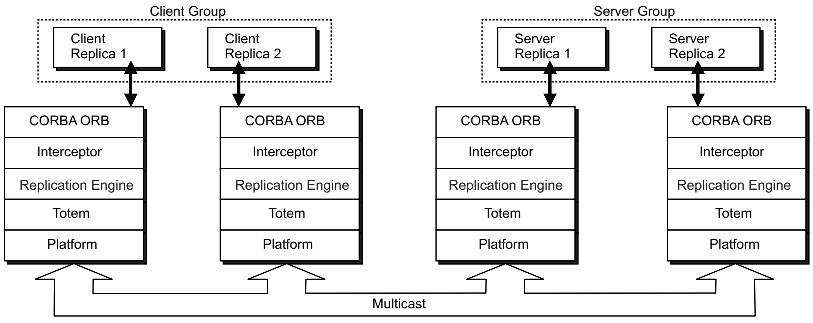


Fig. 1. The components of Eternal's FTORB

The FTORB components interwork as follows. A client group, inside the fault tolerance domain, issues a request to a server group inside the same fault tolerance domain. The Interceptor Library intercepts the request and passes it to the Replication Engine. The Replication Engine receives the request and, instead of forwarding it to the designated server via TCP/IP, forwards the request to Totem, which multicasts it to the server group. The replicas in the server group process the client's request and forward the reply to Totem, which multicasts the reply to the client group. FTORB detects and suppresses duplicate requests and replies from the replicas of the clients and the servers.

2.2 TAO's Load Balancer

TAO's Load Balancer uses a middleware load balancing approach that works with the TAO CORBA ORB [11]. TAO uses the terms Replica Locator and Replica Proxy for two of its modules. If the server that is being load balanced is stateless, those terms are fine. If the server is stateful, the instances of the server that serve different clients are typically not replicas and, thus, we refer to them as peers, instead of replicas. Correspondingly, we use the terms Peer Group, Peer Locator and Peer Proxy for the instances of the server that provide load balancing (see Fig. 2). Use of this terminology avoids confusion between the instances (replicas) of an object that provide fault tolerance and the instances (peers) of an object that provide load balancing.

– **Peer Locator:** The Peer Locator identifies which peer server will receive a client's request and binds the client to the identified peer server. The Peer Locator forwards each request it receives to the peer server selected by the Load Analyzer.
– **Load Analyzer:** The Load Analyzer determines which peer server will receive a client's request and decides when to switch loads between peer servers.
– **Load Monitor:** For a given peer server, a Load Monitor monitors the load on that peer, reports peer loads to the Load Analyzer and responds to load advisory messages from the Load Analyzer.
– **Peer Proxy:** Each object managed by the Load Balancer communicates with the Load Balancer via a Peer Proxy. The Load Balancer uses the Peer Proxies to distinguish between different peer servers.
– **Load Balancer:** A collective term for all of the above components.

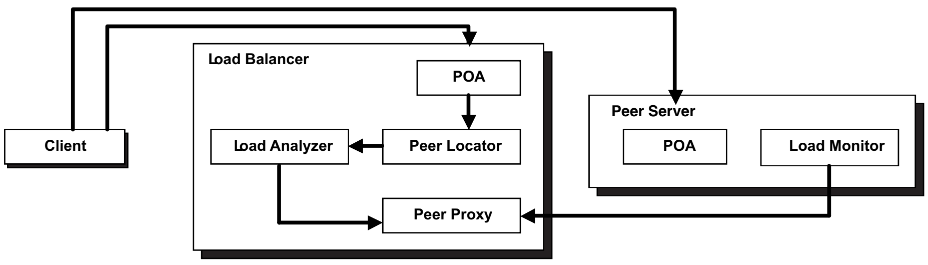


Fig. 2. The components of TAO's Load Balancer

The TAO Load Balancer components interwork as follows. A client obtains an object reference to what appears to be a peer server and issues a request. In actuality, the client invokes the request on the Load Balancer. The Portable Object Adapter (POA) of TAO dispatches the request to the Peer Locator. The Peer Locator queries the Load Analyzer for an appropriate peer server and sends back a LOCATION_FORWARD to the client to redirect the client to the selected peer server. From then on, the client sends its requests directly to that peer server. The Load Analyzer continuously communicates with the Load Monitor for the peer server. If the Load Analyzer finds the peer server to be overloaded, it issues a load advisory to the Load Monitor for that peer server. On receiving the load advisory from the Load Analyzer, the Load Monitor issues a message to the peer server, telling it to accept or redirect the next request.

2.3 Integration of Eternal's FTORB and TAO's Load Balancer

Fault tolerance and load balancing are two orthogonal aspects of quality of service. We consider the composition of these two non-functional properties and the product of peer groups (used for load balancing) and replica groups (used for fault tolerance) in the integrated infrastructure, as shown in Fig. 3.

A peer group, maintained by the Load Balancer, consists of one or more peer servers. Different peer servers handle different clients' requests. The Load Balancer distributes the requests of the different clients across the peer servers in a peer group, in order to balance the load across the peer servers. Each peer server in a peer group has one or more replicas for fault tolerance.

A replica group, created by the FTORB fault tolerance infrastructure, consists of one or more replicas that provide protection against faults. Each peer server is a member of a distinct replica group. With active replication, the replicas of a particular peer server handle the same clients' requests and have the same load.

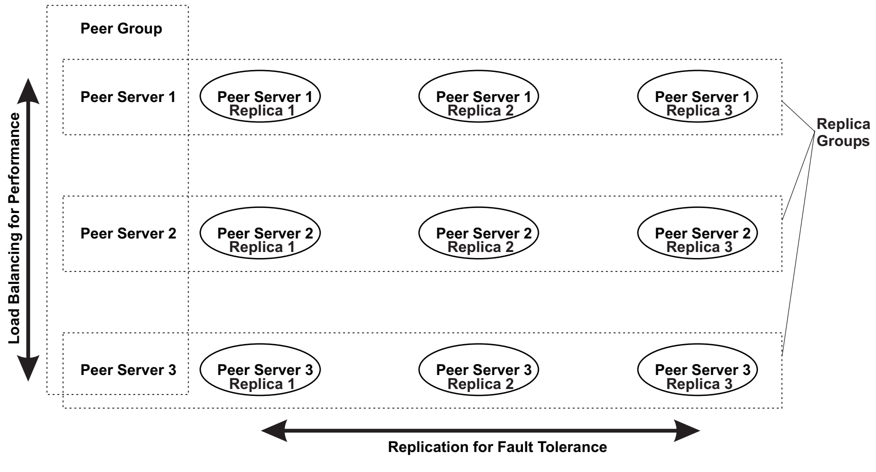


Fig. 3. The product of the peer groups, used for load balancing, and the replica groups, used for fault tolerance, in the integrated infrastructure

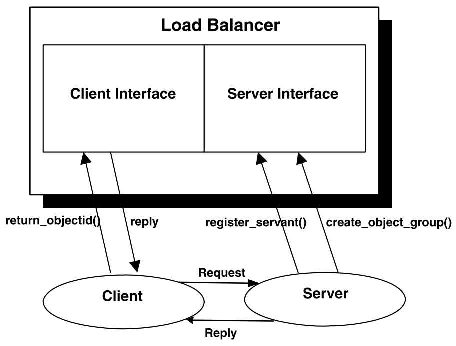


Fig. 4. Interfaces of our Load Balancer

In the integrated infrastructure, not only the peer servers but also the Load Balancer and the clients can be replicated using Eternal's FTORB to provide fault tolerance, so as to avoid a single point of failure. If, during the execution, one of the replicas fails, another replica continues the operations. FTORB then obtains the application state from an existing replica (*i.e.*, takes a checkpoint), and supplies that state to a new or recovering replica, in order to maintain the required level of redundancy.

When a stateful server serves multiple clients, and maintains different state or shared state between them, static load balancing (*i.e.*, directing a client's requests to a particular peer server for an entire session) is appropriate. Dynamic load balancing (*i.e.*, transferring a client's connection from one peer server to another in the midst of a session) is more challenging, because the state of the particular client held by the peer server (rather than the entire state of the peer server) must be transferred from one peer server to another. Our infrastructure provides fault tolerance for stateful servers and uses static load balancing.

In our infrastructure, the Load Balancer has two interfaces, the Server Interface and the Client Interface, as shown in Fig. 4 and described below.

- **Server Interface:** The server interface exposes methods that are invoked by the servers. These methods include *create_object_group()* and *register_servant()*, which enable the peer servers to create a peer group and register with the Load Balancer. After the peer servers have created a peer group and registered with the Load Balancer, the Load Balancer has a clear view of the peer servers that are available. The Load Balancer is then ready to balance the loads across the peer servers within the peer group, on receiving requests from the clients.
- **Client Interface:** The client interface exposes methods that are invoked by the clients. These methods include the *return_objectid()* method, which enables a client to obtain an object group reference for a peer server's replica group. On receiving a request from a client for an object group reference corresponding to a particular object id, the Load Balancer looks in its table to find an appropriate reference, using its particular load balancing policy to select a peer server, and then returns the object group reference of the peer server's replica group to the client. On obtaining the reference, the client issues requests to the peer server, which are multicast to the peer server's replica group.

Fig. 5 shows the use of our infrastructure in an example configuration consisting of the Load Balancer with two replicas each, two peer servers with two replicas each, and three clients with one replica each. The dashed boxes represent the replica groups. Each request (reply), instead of being sent to a single server (client), is multicast to a server (client) replica group. For each peer group, the load balancing takes place across the two peer servers that constitute that peer group.

First, the replicas of the Load Balancer are brought up on different processors. The first replica invokes FTORB to create a replica group with itself as a member, and then the second replica adds itself to the replica group.

Next, the replicas of the peer servers are brought up on different processors. Each peer server invokes FTORB to create a replica group with itself as a member and registers with the Load Balancer, using a multicast request (arrows 1 and 2). The Load Balancer creates a peer group, and adds each peer server to the peer group. The subsequent replicas of each peer server add themselves to the replica group of that peer server.

Then, the clients are brought up on different processors. Each client invokes FTORB to create a client replica group, consisting of one member each. In the example, client 2 invokes the Load Balancer's *return_objectid()* method to request a reference for a peer server, using a multicast request (arrow 3). Depending on its load balancing policy, the Load Balancer replies with the object group reference of the replica group of one of the peer servers, using a multicast reply (arrow 4), in this case, peer server 1.

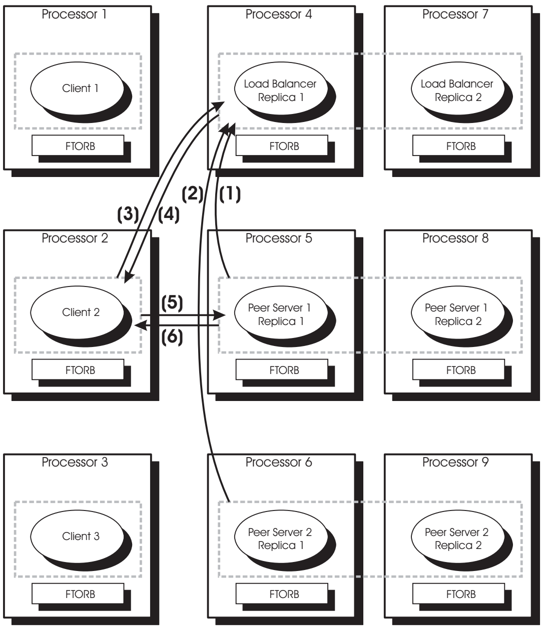


Fig. 5. The Load Balancer and the two peer servers are actively replicated, with two replicas each, using FTORB to provide fault tolerance. For each peer group, the load balancing takes place across the two peer servers that constitute that peer group

After receiving the peer server reference, client 2 invokes methods directly on peer server 1, using the object group reference of that peer server's replica group and a multicast request (arrow 5). The replicas of peer server 1 generate the response and multicast the reply to client 2, using a multicast reply (arrow 6).

3 Challenges and Their Solutions

Now we discuss the challenges that we faced in designing and implementing the integrated infrastructure and our solutions to those challenges. The challenges can be divided into two categories:

– Challenges faced in designing the Load Balancer
– Challenges faced in designing the client and server applications.

3.1 Challenges in Designing the Load Balancer

First we discuss challenges that we faced in designing the Load Balancer.

Challenge 1: TAO's Load Balancer provides load balancing by intercepting the clients' requests and returning a LOCATION_FORWARD, which contains the reference to the selected peer server. When the client receives the LOCATION_FORWARD, it sends its subsequent requests to that peer server.

To intercept requests, the Load Balancer uses the Servant Manager in a Portable Object Adapter (POA) that has USE_SERVANT_MANAGER and NON_RETAIN policies. Servant Managers are used to activate servants dynamically. Servant Managers have two interfaces, the ServantActivator and ServantLocator interfaces. TAO's Load Balancer uses the ServantLocator interface to intercept a client's request and instantiate a servant for it on the fly.

This approach works fine for load balancing, but does not work when integrated with fault tolerance. The reason is that Eternal's FTORB recognizes replicas only if they have the same object ids. Because the POA's policy is set to NON_RETAIN, the object ids of the activated server objects are not persistent and, hence, are not recognized as replicas by FTORB.

Rejected Strategies: It was not possible to solve this problem by changing the POA's policies, because the USE_SERVANT_MANAGER and NON_RETAIN policies were required to use the ServantLocator interface.

First, we considered dividing the functionality of TAO's Load Balancer into two parts. One part would interact with the servers and the other part with the clients. This solution was not feasible for the reason that it required that one POA exposes methods for the servers and has the PortableServer::PERSISTENT and PortableServer::USED_ID policies, and the other POA exposes methods for the clients and has the USE_SERVANT_MANAGER and NON_RETAIN policies. However, the POA with the ServantLocator interface is unaware of server objects that are active inside the other POA. Moreover, this solution would misuse the ServantLocator, as servant objects are already active and, thus, do not require the services of the ServantLocator.

Another possible solution was to write our own interceptors using CORBA's portable interceptors, which replace the ServantLocator interface in the Servant Manager. Portable interceptors expose pre-defined interception points, at both the client and the server, which make the requests and replies accessible. In TAO's Load Balancer, we intercepted the client's request, so the interception points are *receive_request_service_contexts()* and *receive_requests()* on the server-side. The problem was that, when the request was intercepted at the *receive_request_service_contexts()* interception point, we did not have access to the object id, as operation parameters were not available. Thus, it was impossible to determine the server group to which the request was addressed. Before the request reached the *receive_requests()* interception point, where the parameters were accessible, it was too late. The reason is that, after *receive_request_service_contexts()*, the servant manager was invoked, but as the servant manager was not activated, a system exception was raised. Thus, using CORBA's portable interceptors did not work.

Solution: The solution to this problem is to have the clients and servers communicate directly with the Load Balancer. The clients and servers are responsible for resolving the reference to the Load Balancer, and issue explicit requests to it. By adopting this solution, we avoided using both the ServantLocator and CORBA's portable interceptors.

In this approach, the servers are activated inside a POA, having the PortableServer:: PERSISTENT and PortableServer:: USED_ID policies. Because these policies are set, the object group references, which are written to IOR files, are persistent. As their object ids are persistent, FTORB can recognize the server replicas. The client does not send its first request to the Load Balancer, but resolves the reference to the Load Balancer and queries the Load Balancer for the peer server to which to connect. In the previous approach, the Load Balancer returns the object group reference for the peer server's replica group to the client, which also happens with this approach. The difference is that now the client explicitly requests the object group reference for the peer server's replica group from the Load Balancer.

Challenge 2: Eternal's FTORB uses two methods, *get_state()* and *set_state()*, for checkpointing and recovery. When a new replica is added to a group, FTORB invokes the *get_state()* method of an existing replica to obtain the state of that replica. It then invokes the *set_state()* method of the new replica to initialize its state and replays the messages from the message log. The new replica begins processing requests, from that point onwards in the message sequence.

The Load Balancer is a stateful server and, thus, the challenge is to identify the state of the Load Balancer and to implement the *get_state()* and *set_state()* methods for it.

Solution: We identified the state of the Load Balancer, in particular the state that was stored in its table when the peer servers registered with the Load Balancer and that associate the object ids of the peer servers with their object group references. The Load Balancer uses this state to identify different peer servers, select a peer server to process a client's request, and supply object group references to the clients.

We then coded the *get_state()* and *set_state()* methods for the Load Balancer, which retrieve the state from one replica of the Load Balancer and set it within another replica of the Load Balancer, respectively.

3.2 Challenges in Designing the Client and Server Applications

Next we discuss some of the challenges that we faced in designing the client and server applications.

Challenge 3: The *get_state()* and *set_state()* methods must be coded for the client and server applications to make them fault tolerant, as described above. The challenge is to identify the state of the client and server applications.

Solution: For each client and server application in our examples, we identified the state and coded the *get_state()* and *set_state()* methods for that client and server. Global variables and data structures that are retained from one request to the next must be recorded, but local variables, such as loop indices, need not be.

Challenge 4: Only one peer server per peer group invokes the *create_object_group()* method of the Load Balancer to create a peer group. Once the Load Balancer has created the peer group, the other peer servers first obtain the object group reference of the peer group and then invoke the *register_servant()* method to add themselves to the peer group. TAO's Load Balancer does not provide this functionally. The challenge is to incorporate it within our integrated infrastructure.

Solution: One possible solution to this challenge was to recode the method *create_object_group()* in the TAO Load Balancer. Another possible solution was to make the necessary changes within the server to obtain the object group reference of the peer group that was created.

Because *create_object_group()* is part of TAO's Load Balancer, we adopted the second solution. Thus, whenever we create a peer group, we write its object group reference to an IOR file. Other peer servers that wish to add themselves to the peer group read the object group reference from the IOR file.

4 Performance Measurements

We describe three of the experiments that we performed for the integrated infrastructure and give the corresponding performance measurements.

The experiments were performed on up to nine Pentium III 1 GHz computers, connected by a 100 Mbps Ethernet switch. The computers ran the Linux Red Hat 8.0 operating system and the TAO CORBA ORB [11].

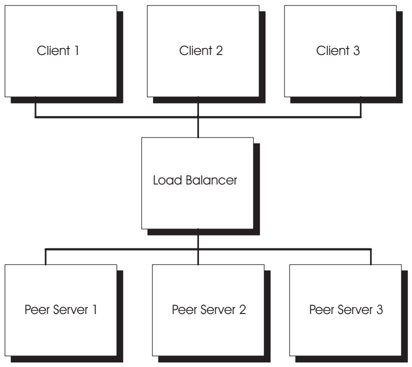
The applications consisted of a client that invokes a server remotely across the network. The client incurs a random delay (think time) between requests. The server performs a nominal processing operation of 30000 microseconds before responding to the client. Request and reply messages are 1kByte each.

We measured the response time seen by a client (*i.e.*, the time interval in microseconds between a client's issuing a request and receiving a reply from the server) and the throughput of a peer server (*i.e.*, the number of requests per second handled by the peer server).

4.1 Experiment 1: Decrease in Response Time with Load Balancing

Fig. 6 shows the testbed for this experiment, which consists of the Load Balancer, three clients and three peer servers, each having one replica. Each of the objects ran on a different processor, with FTORB running on all of the processors. First, we performed the experiment with one peer server serving the three clients. Next, we performed the experiment with three peer servers serving the three clients.

The graph in Fig. 6 shows the two configuration setups on the horizontal axis and the response time for a client's request on the vertical axis. Point A represents the setup with one peer server serving the three clients, and point B represents the setup with three peer servers serving the three clients. The experimental results show that the response time improved by 32%, when the clients' requests are load balanced across the three peer servers.



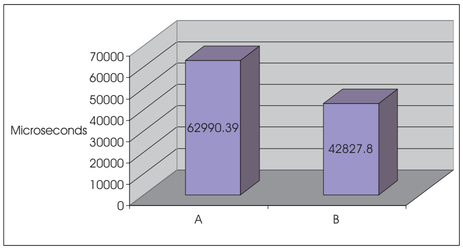


Fig. 6. Experiment 1. Testbed setup and response time for a client's request with load balancing: (A) one peer server and (B) three peer servers

4.2 Experiment 2: Increase in Throughput with Load Balancing

Fig. 7 shows the testbed for this experiment, which consisted of the Load Balancer, five clients and two peer servers, each having one replica. Each of the objects was hosted on a different computer, and FTORB was running on all of the computers.

The graph in Fig. 7 shows the server throughput curves without load balancing (bottom curve) and with load balancing (top curve). The vertical axis represents the throughput in requests per second. The horizontal axis represents decreasing random delays (think times) between requests at the clients. Initially, both of the throughput curves increase and there is no queuing at the peer server. After the peer server reaches its maximum processing capacity, the throughput curves flatten out and arriving requests from the clients are queued.

With load balancing across the two peer servers, we observed that the clients are distributed in groups of two and three between the two peer servers. This division resulted in higher throughput and decreased load on the single peer server without load balancing. When we decreased the number of clients communicating with the single server without load balancing from five to three, the throughput of the server increased by 15%. Thus, integrating load balancing resulted in a 15% increase in throughput.

4.3 Experiment 3: Increase in Response Time with Replication

Fig. 8 shows the testbed for this experiment, with the Load Balancer, one client and one peer server, having one, two or three replicas. Each of the objects ran on a different pro-

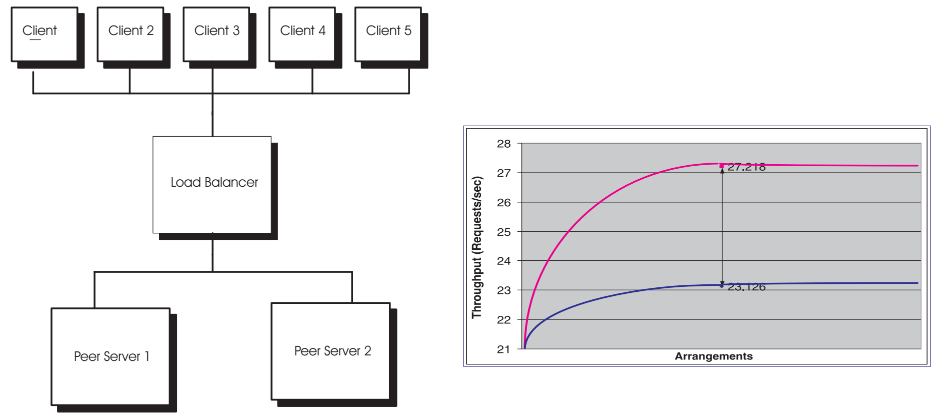


Fig. 7. Experiment 2. Testbed setup and throughput of a peer server: (A) bottom curve without load balancing and (B) top curve with load balancing

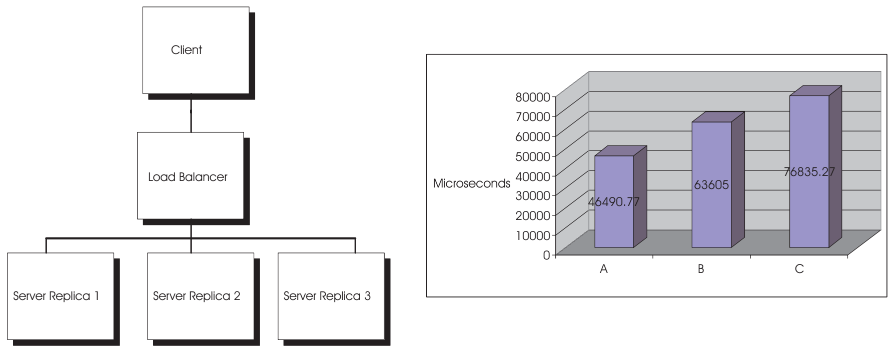


Fig. 8. Experiment 3. Testbed setup and response time: (A) one server replica, (B) two server replicas and (C) three server replicas

cessor, with FTORB running on all of the processors. In this experiment, we investigated the response time as the number of replicas of a peer server is increased.

The graph in Fig. 8 shows the results of the experiment. The vertical axis shows the response time in microseconds. Points A, B and C on the horizontal axis represent the three configurations when one, two and three server replicas are running. The results show that there is a 37% overhead when the number of replicas is increased from one to two and a 20% overhead when the number of replicas is increased from two to three. The primary cause of the increased overhead is the Totem multicast protocol, the latency of which increases linearly with the number of processors on the ring [12].

5 Related Work

Research on load balancing for distributed systems typically focuses on dynamic strategies and algorithms that maximize throughput and minimize overhead, and does not also deal with fault tolerance. Likewise, research on fault tolerance for distributed systems focuses primarily on strategies and mechanisms that achieve a high quality of fault tolerance, and does not also address load balancing or other quality of service issues. A few infrastructures have been developed that provide both fault tolerance and load balancing. Generally, they do not provide fault tolerance transparently but, rather, depend on modifications to the application programs.

Beowulf is a parallel processing environment that provides dynamic, on-board, adaptive distribution of processing tasks across a heterogeneous network of processors. It allocates processing to off-board resources as appropriate and as resources become available. In [2] Bennett, Davis and Kunau describe ParaSort, a distributed parallel data allocation sorting algorithm with automatic load balancing and fault tolerance that operates in the Beowulf environment. The fault tolerance is explicitly programmed into the application, in contrast to our more transparent fault tolerance approach.

AQuA [10] is a dependability framework that provides object replication and fault tolerance for CORBA applications. AQuA exploits the group communication facilities and message ordering guarantees of the Ensemble and Maestro toolkits to ensure replica consistency. AQuA supports both active and passive replication, with state transfer to synchronize the states of the backup and primary replicas for passive replication. AQuA also addresses resource management and other quality of services issues, but not load balancing based on the CORBA standard like our infrastructure does.

Ho and Leong [3] have extended the CORBA Event Service with load balancing and fault tolerance in a transparent manner. Their framework replicates event channels and shares the load among the replicas, using both static and dynamic load balancing, to improve scalability. It monitors the event channel replicas and, if an event channel replica becomes faulty, it transfers the consumers of the faulty event channel replica to another event channel replica and restarts the faulty event channel replica. Unlike our infrastructure which replicates stateful application objects, their framework replicates stateless event channels.

JBoss [4] is an open-source Java EJB/J2EE application server and, as such, it uses CORBA's IIOP protocol. JBoss has been extended with the JavaGroups group communication toolkit [1], a Java implementation of the Ensemble toolkit, to provide clustering, including load balancing, session state replication, and failover. JBoss uses an abstraction framework to isolate communication layers and, thus, like our infrastructure, achieves transparency to the applications and other middleware.

Petri, Bolz and Langendorfer [9] have developed a system that provides load balancing and fault tolerance for compute-intensive scientific applications. Their system uses a global virtual name space for groups of processes distributed across a workstation cluster. Applications use the same virtual names for operating system objects, independent of their location. System calls are interposed via the debugging interface, and parameters are translated between name spaces. Thus, like our infrastructure, their system uses library interpositioning to achieve transparency to the applications.

6 Conclusion

We have presented an integrated software infrastructure, based on Eternal's FTORB and TAO's Load Balancer, that renders distributed applications, based on CORBA, both fault tolerant and load balanced. We have discussed challenges that we faced in the integration and solutions to those challenges. We have also presented performance measurements, which show that the integrated infrastructure resulted in some overhead, as one would expect. However, the integrated infrastructure increases the robustness of the services that the applications provide to the clients and results in improved response time for the clients and throughput of the servers. Our work has shown that integrating middleware components, such as TAO's Load Balancer and Eternal's FTORB, that provide orthogonal non-functional properties might require modifications to one or both components, because each makes assumptions that the other might not satisfy.

References

1. B. Ban, "Design and implementation of a reliable group communication toolkit for Java," (September 1998), *http://www.cs.cornell.edu/home/bba/Coots.ps.gz*.
2. B. H. Bennett, E. Davis and T. Kunau, "Beowulf parallel processing for dynamic load balancing," *Proceedings of the IEEE Aerospace Conference,* vol. 4 (March 2000), Piscataway, NJ, pp. 389-395.
3. K. S. Ho and H. V. Leong, "An extended CORBA event service with support for load balancing and fault tolerance," *Proceedings of the IEEE International Symposium on Distributed Objects and Applications* (September 2000), Antwerp, Belgium, pp. 49-58.
4. B. Burke and S. Labourey, "Clustering with JBoss 3.0" (July 2002), *http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html*.
5. L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos, "Totem: A fault-tolerant multicast group communication system," *Communications of the ACM*, vol. 39, no. 4 (April 1996), pp. 54-63.
6. P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Strongly consistent replication and recovery of fault-tolerant CORBA applications," *Computer System Science and Engineering Journal*, vol. 17, no. 2 (March 2002), pp. 103-114.
7. Object Management Group, Fault Tolerant CORBA, OMG Technical Committee Document ptc/2000-04-04 (April 2000).
8. Object Management Group, CORBA Load Balancing and Monitoring Specification, OMG Document mars/02-10-14 (October 2002).
9. S. Petri, M. Bolz and H. Langendorfer, "Migration and rollback transparency for arbitrary distributed applications in workstation clusters," *Proceedings of the Parallel and Distributed Processing Symposium* (March-April 1998), Berlin, Germany, pp. 159-170.
10. Y. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M.Seri, "AQuA: An adaptive architecture that provides dependable distributed objects," *IEEE Transactions on Computers*, vol. 52, no. 1 (January 2003), pp. 31-50.
11. D. C. Schmidt, D. L. Levine and S. Mungee, "The design of the TAO real-time object request broker," *Computer Communications*, vol. 21, no. 4 (April 1998), pp. 294-324.
12. E. Thomopoulos, L. E. Moser and P. M. Melliar-Smith, "Latency analysis of the Totem single-ring protocol," *ACM/IEEE Transactions on Networking*, vol. 9, no. 5 (October 2001), pp. 669-680.
13. W. Zhao, L. E. Moser and P. M. Melliar-Smith, "Unification of replication and transaction processing in three-tier architectures," *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Vienna, Austria (July 2002), pp. 290-297.

Performance Evaluation of Consistent Recovery Protocols Using MPICH-GF

Namyoon Woo[1], Hyungsoo Jung[1], Dongin Shin[1], Hyuck Han[1], Heon Y. Yeom[1], and Taesoon Park[2]

[1] School of Computer Science and Engineering,
Seoul National University
Seoul, 151-742, Korea
{nywoo, jhs, dishin, hhyuck, jhs, yeom}@dcslab.snu.ac.kr
[2] Department of Computer Engineering,
Sejong University, Seoul, 143-747, Korea
tspark@sejong.ac.kr

Abstract. This paper presents an implementation of several consistent recovery protocols at the abstract device level and their performance comparison. We have performed experiments using three NAS Parallel Benchmark applications with class C datasets on state of the art equipment. The interesting result is that causal message logging protocol has the most expensive recovery cost with communication intensive applications since it suffers from concentrated overload of simultaneous message replaying. Receiver-based optimistic message logging has the least recovery cost with drawback of extensive disk access overhead in failure-free executions. Coordinated checkpointing seems the most practical choice among them.

1 Introduction

Fault-tolerance issues are gaining considerable attention from cluster researchers as the scale of distributed system increases since the distributed systems are not so reliable to guarantee the completion of jobs in a determinate time for their inherent failure factors. Even a single local failure could nullify all of the computation mid-results, so it is indispensable to provide fault-tolerance in order to increase the reliability of the whole system.

Checkpointing/rollback-recovery is a well-known fault-tolerance technique for parallel processes. Checkpointing is an operation to store the states of processes into the stable storage for the purpose of recovery or migration. Periodic checkpointing minimizes the computation loss incurred by failures. For parallel processes to recover from failures, it is important to keep the consistency among them since their states have causal relationship with one another by exchanging messages [5]. Over two decades, many researches have conducted to provide consistent rollback-recovery algorithms for parallel processes [6]. Recovery algorithms are categorized mainly into three groups: coordinated checkpointing, communication-induced checkpointing or message logging. Coordinated checkpointing protocols force all processes to coordinate before checkpointing, so that every global checkpoint is consistent. Communication-induced checkpointing allows a process to checkpoint independently, but also forces a process to checkpoint if some conditions are satisfied to prevent useless checkpoints. The main drawback is that it could generate large number of forced checkpoints unpredictably [1]. Message logging protocols allow each process to checkpoint independently with recording all the messages exchanged. When a process fails accidentally, it restores its state by rolling back to the latest checkpoint and by replaying all the messages in the same order as they were processed before. Message logging protocols can be categorized further into sender-based, receiver-based, or causal message logging.

There have been several efforts to make theories practical and to compare these protocols [4, 10, 11, 15, 17, 18, 20]. The previous works employed some subset of recovery protocols, and evaluated their performance using applications running on small datasets. Most of them agree that coordinated checkpointing performs better than message logging, but still doubt its scalability. They recommend message-logging protocols for fast recovery. However, it is still unsure which message logging protocol is superior to the others. The purpose of this article is to evaluate the performance of consistent recovery protocols in practical environment.

This paper describes how we have implemented the recovery protocols and how they perform with applications of large datasets. We integrate fault tolerance module to Message Passing Interface (MPI) implementation that is the [illegible] standard specification for parallel programming [7]. Our framework, MPICH-GF is a fault-tolerant MPI implementation, which has been developed for the use under grid. MPICH-GF provides total user-transparency so that application codes do not have to be re-written or users do not have to be aware of its existence. Current MPICH-GF version employs three protocols: coordinated checkpointing, receiver-based optimistic message logging, and causal message logging. We measure their performance by running NAS Parallel Benchmark suites (NPB) [9] with class C datasets. Our observation is that message logging protocols do not seem practical for their excessive resource consumption. The recovery of causal message logging is not as fast as expected. Even with the enhanced industrial technology in disks, checkpointing into disks is still the dominant source of total overhead.

The rest of this paper is organized as follows. In Section 2, we present the related works in fault-tolerance area. Section 3 introduces MPICH-GF system that is the framework in this paper. Section 4 describes how we have implemented consistent-recovery protocols, and how we have modified the original protocols. The experimental results are shown in Section 5. We conclude this paper in the final section.

2 Related Works

RENEW [10] is the fault-tolerant MPI that supports coordinated checkpointing, communication-induced checkpointing, and sender-based message logging. They conclude that while sender-based message logging shows poor performance on failure-free execution, all protocols have the almost same recovery costs. Egida [16] researchers have shown the performance comparison of some message logging protocols in their literature [15]. Their conclusion is that receiver-based message logging recovers faster than sender-based logging especially on concurrent failures, but it faces a complex implementation task. The experiments in RENEW and Egida have been performed with only four processes.

Manetho [20] researchers assert that logging messages at the sender is preferred to logging at the receiver and that the cost of running the recovery protocol is negligible in comparison. They performed tests with small datasets where the sizes of checkpoint files were at most 37 Mbytes and the communication rates were low.

MPICH-V2 [4] is the most recent fault-tolerant MPI implementation that supports coordinated checkpointing and sender-based pessimistic message logging. The unique feature is that it uses remote event loggers that store message reception orders. Since checkpoint files are stored at the remote storage server, coordinated checkpointing suffers from network contention on checkpointing as well as on recovery. In their literature, they assert that sender-based message logging is recommendable on the system of high failure frequency, since sender-based message logging has the small recovery cost.

CoCheck [18] is a thin library over PVM, supporting coordinated checkpointing. Legion MPI-FT [11] is the first effort to build the fault-tolerance system on the grid. It also supports a coordinated checkpointing protocol. Starfish [3] is a heterogeneous checkpointing toolkit based on Java virtual machine, which enables processes to migrate among heterogeneous platforms. Hector [17] exists as a movable MPI library and several executables. It supports coordinated checkpointing; before checkpointing, every process closes its channel connection to ensure that there are no in-transit messages left in the network. Processes have to reconstruct their channels after checkpointing.

3 MPICH-GF

Our framework consists of MPICH-GF library and a few executables. MPICH-GF is a fault-tolerant MPI implementation of our own, which originates from MPICH-G2 [8]. MPICH-G2 uses a lightweight abstract device that operates on grid architectures[1]. MPICH-GF is completely user-transparent, so that application codes do not have to be re-written or the users need not be aware of its existence.

3.1 MPICH-GF Library

Figure 1 describes the MPICH-GF library structure. It contains a checkpoint toolkit, an atomic message transfer module, a channel re-establishment mod-

[1] This research is part of Korean National Grid Projects that require participating researchers to agree to MPICH-G2. However, we believe that our methods in building fault tolerance module would be valid still with *ch_p4* device, since the areas above the atomicity of message transfer implementation are fundamentally same.

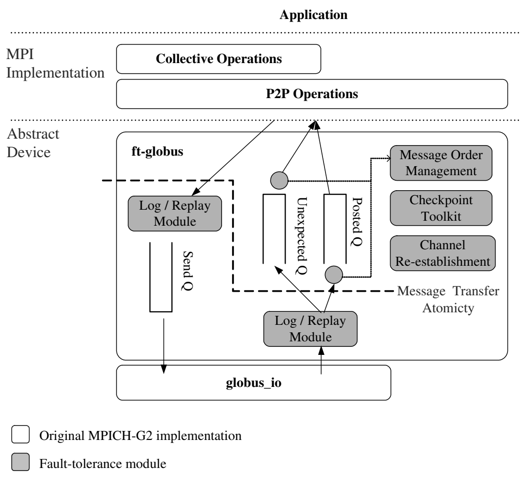


Fig. 1. The architecture of MPICH-GF library

ule, and a message logging / replaying module. MPICH upper layer contains blocking/non-blocking point-to-point (p2p) operations as well as collective operations that exist on p2p operations. In the abstract device, a blocking operation is a combination of a non-blocking operation and a probe operation. Message transfer of MPICH-GF is polling-driven. Our device is based on the TCP socket so that FIFO property is kept for channels. There are two kinds of receive queues in the device; the [illegible] contains receive requests of which the corresponding messages have not arrived yet, while the [illegible] contains received messages of which the corresponding requests from the upper layer have not been issued yet.

As a checkpoint toolkit, we adopted Zandy's checkpoint library [19] that dumps the user-level memory of a target process into the file system. On recovery, some processes may have to rollback to the previous checkpoint for consistency, in which case each process overwrites its user memory with the corresponding checkpoint file. However, channels in kernel are still available, so the processes that rolled back do not have to re-initialize their channel information.

The thick dashed line in the figure stands for the atomicity of message transfer, which means the message transfer operation is mutually exclusive to the checkpointing procedure. This atomicity is for two purposes; one purpose is to keep the message state either 'completed' or 'not-delivered at all' in checkpoint files since it is troublesome on recovery to handle the messages that are recorded as in-transit in checkpoint files. The other purpose is to prevent message transfer codes from being interrupted by checkpointing request. The interrupted message transfer codes may be re-entered by recovery protocols for sending coordination messages or for replaying message logs, which is not safe at all. In MPICH-GF, if the messages are under delivery, the request for checkpointing is delayed until the delivery completion. We have designed code-areas that should be exclusive as narrow as possible for fine-grain checkpointing. The mutually exclusive operations are the non-blocking send ([illegible]), and send/recv probe operations ([illegible]).

3.2 Hierarchical Job Managers

Figure 2 describes how managers launch and manage MPI processes on the globus middleware. There are three main components: a central manager, a cluster manager, and a local manager. We have implemented the central manager in the DUROC (Dynamic Updated Request Online Co-allocator) component of Globus toolkit that handles job-submission, job-monitoring, and channel initialization for MPI processes. We have added the dynamic process management for failed MPI processes, and the management of consistent recovery.

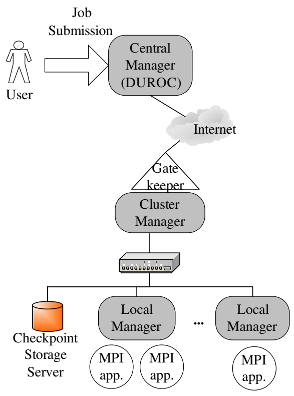


Fig. 2. The hierarchical job manager system

The cluster manager relays control messages between the central manager and local managers in order to moderate the communication overhead concentrated on a single point. It detects and handles node failures with a heartbeat technique. The cluster manager runs at the front end of cluster system. Therefore, there are as many cluster managers as the number of clusters.

Each local manager forks and monitors a MPI process. It receives a SIGCHLD signal when its forked process terminates, then it checks with the help of [illegible] *()* system call whether the termination was through normal [illegible] *()* system call. If not, it regards the event as a process failure, and reports the failure event to the central manager.

4 Implementation

4.1 Coordinated Checkpointing

Our coordinated checkpointing protocol is straightforward: The central manager initiates global checkpointing periodically, then local managers signal processes to be ready for checkpointing. On receipt of the signal, the signal handler in an MPI process executes a barrier-like function before checkpointing. This barrier procedure guarantees two things: One is that any pair of processes is not dependent on each other, and the other is that no messages exist in network because barrier messages push all the previously issued messages. After barrier, each process generates its checkpoint file, and informs the local manager of the successful completion. The central manager confirms a new version of global checkpoint after collecting all completion reports. In the sense that barrier messages are used, our protocol may look like [illegible] style [5], but barrier messages do not play the role of checkpoint-request. It is rather a kind of blocking coordinated checkpointing protocol.

On failure, the central manger determines what version of global checkpoint to use for recovery, and requests all the other survived processes to rollback. The recovered process rejoins the MPI process group by informing its new channel information. After all processes get the new channel information of the recovered process, the computation resumes.

4.2 Causal Message Logging

Our causal message logging is based on Family-based logging protocol [2] where the message body is logged at the sender's memory. The receiver generates the causality information - determinants -, and then piggybacks them on the next sent message. The determinant consists of three integers: a send sequential number, a reception sequential number, and a global rank of the sender process.

In causal message logging(CML), checkpoint files include message logs, so CML has larger size of checkpoint files than the checkpoint size of the other protocols. After checkpointing, each process performs a garbage collection procedure asynchronously in order to remove old logs from memory.

The recovery procedure is as follows: (1) After rejoining, (2) the recovered process broadcasts its causal relation status. The other processes also report the latest message ID received from the recovered process, and send the corresponding determinants to the recovered process. (3) The recovered process collects determinants, and determines the delivery order of replayed messages. The survived processes re-send the message logs, if any, to the recovered process. (4) During re-execution of the lost computation, the recovered process does not send any messages that are sent already in the previous instance.

MPICH-GF does not record the reception order for the messages that correspond to the receive function calls with the specific sender ID, since they are deterministic events. However, an application may specify `MPI_ANY_SOURCE` option in a receive function call, which is bound to the earliest message, so it is non-deterministic. MPICH-GF records the message delivery order for receive functions with `MPI_ANY_SOURCE` option only. Then, how can we know whether a message would be bound to a receive function with `MPI_ANY_SOURCE` or not? It is determined at the queue lookup module as shown in Figure 1: when a receive request looks up the [illegible] and when an arrived message looks up the [illegible] for the matching request. If this logging management exists out of the abstract device, like MPICH-V2 [4], it would be impossible to tell whether a message would be delivered in a deterministic order or not. In such a case, there is no choice but to record all messages' arrival. On recovery, if the recovered process calls a receive function with `MPI_ANY_SOURCE` option, and if there is the corresponding information of delivery-order, MPICH-GF replaces `MPI_ANY_SOURCE` with the rank of the sender process. The receive request is bound to the exactly same message as it was in the previous execution.

4.3 Optimistic Message Logging

In our optimistic message logging (OML), the message body is stored at the receiver's disk asynchronously on its arrival. Each message log is stored sequentially in its arrival order. While the procedure in failure-free execution is simple, the recovery procedure is rather complex. In the previous literatures, the recovery protocol operates in a distributed way, and a cascade of rollbacks may happen to some degree for the lost state. We want to do the whole recovery procedure in one phase in order to simplify the procedure, so we adopt the centralized way. On recovery, each process reports its causal relationship status to its manager, the central manager collects them, calculates the recovery line, and determines which processes to rollback. Recovered/rolled-back processes read the message logs from disk, and put them into the receive queue. Finally, the recovered process rejoins into the process group. We place rejoining at the last in the recovery procedure in order to finish the message replay before the other processes send normal messages.

5 Performance

5.1 Experimental Conditions

All experiments have been performed on a 32-node cluster where each node is equipped with dual Intel Xeon processors running at 2.4 GHz, with 2 GB main memory, and a 140 GB SCSI hard disk. A Gigabit network switch connects all nodes. The operating system is Linux 2.4, and Globus toolkit v2.2 is installed at the front node.

Table 1. Characteristics of the NPB applications used in the experiments

Appl. name (iter.)	Comm. pattern	Num. of proc.	Memory Usage /proc (MB)	Total Num. of received mesg.	Num. of MPI_ANY_SOURCE	Total mesg. size (MB)	Num. of ckpts.
bt.c (200)	3D cube	9	477	3666	0	1513.4	5
		16	263	4885	0	1265.9	
		25	173	6101	0	1089.9	
cg.c (75)	Multiple chains	8	155	13999	0	1696.1	5
		16	89	14004	0	1696.1	
		32	48	20013	0	939.0	
lu.c (250)	2D mesh	8	101	120796	759	939.0	10
		16	55	161051	1010	666.2	
		32	32	161063	1010	463.5	

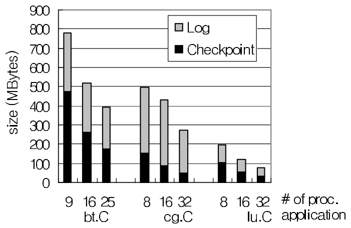


Fig. 3. The size of single checkpoint and message logs within a checkpoint period

To evaluate the performance of each protocol, we ran the experiment using three applications of the NAS Parallel Benchmark suite [9] with class C dataset: BT, CG, and LU. Their execution times are a few tens of minutes, which are adequate for our repeated experiments. [2] The profile of each application is shown in Table 1. BT solves Navier-Stokes equation consuming the largest memory among three applications. CG is a conjugate gradient method, which has the largest total message size. LU decomposition application consumes relatively small amount of memory, and exchanges many small messages. Only LU issues receive function calls with `MPI_ANY_SOURCE` option, which are about 0.6% of the total number of receive calls. Therefore, the total amount of determinants piggybacked on messages by causal message logging is negligible.

Figure 3 shows the size of single checkpoint and the size of accumulated message log within a checkpoint period. Specially, the checkpoint size of causal message logging is the sum of checkpoint size and log size in the figure. Since the checkpoint toolkit dumps the user memory that a target process occupies, the size of checkpoint file is the same as the amount of used memory. In all cases, the memory usage does not exceed the size of the equipped main memory (2GB), so there have been no swapping effects in our experiments. The memory usage decreases as the more processes participate, since the dataset are more distributed in proportion to the degree of parallelism. Checkpoint files are stored at the local disk of each node.

5.2 The Overhead in Failure-Free Execution

Figure 4 compares the total execution time for each application with coordinated checkpointing (CC), receiver-based optimistic message logging (OML), causal message logging (CML), and without any consistent recovery protocol (none). Coordinated checkpointing shows the best performance among three protocols as expected. The communication frequency and the message size influence the performance of logging protocols significantly. The performance difference of OML and CML is small with BT application that has the smallest frequency

[2] The execution times of NPB with class D were from an hour to days on the system that we rented. It was impossible to complete the experiments with class D datasets in permitted time. Anyhow, the tendency in the results of class C seemed to appear again with class D.

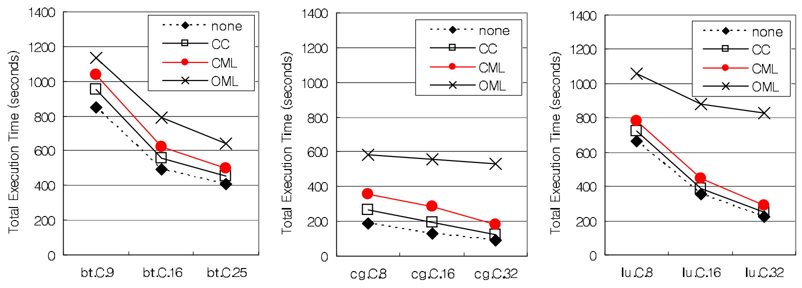


Fig. 4. The comparison of total execution time for consistent-recovery protocols

of communication, while it is large with LU, the most communication intensive application. The reason why OML is inferior to CML is that OML accesses disks more frequently than CML. CML stores the chunk of message log into disk at checkpointing time, while OML stores each message into disk on its arrival. What is worse with OML, the kernel flushes buffers into disks too often even with asynchronous disk access, for the huge message size of class C NPB applications.

Table 2 shows the composition of single checkpointing overhead. The 'channel flush' in the table means the procedure to make un-finished message transfer completed before checkpointing in order to handle in-transit messages. Besides flushing channel, OML completes the asynchronous message logging by flushing the disk write stream before checkpointing ('logging check'.) In CC, it is hard to find any scalability issues on barrier performance. In all cases, disk write operation is dominant in the whole overhead.

Table 2. The composition of single checkpointing overhead (unit = seconds)

Applications		bt.c			cg.c			lu.c		
Protocols		9	16	32	8	16	32	8	16	32
CC	channel flush	0.16 (0.7%)	0.13 (1.0%)	0.13 (1.8%)	0.06 (0.8%)	0.04 (0.9%)	0.06 (3.0%)	0.02 (0.4%)	0.02 (0.7%)	0.02 (1.5%)
	barrier	0.19 (0.9%)	0.20 (1.6%)	0.19 (2.6%)	0.15 (2.0%)	0.10 (2.2%)	0.12 (5.9%)	0.04 (0.8%)	0.09 (3.2%)	0.10 (7.3%)
	disk write	21.43 (98.4%)	12.35 (97.4%)	6.93 (95.6%)	7.48 (97.2%)	4.32 (96.9%)	1.85 (91.1%)	4.90 (98.8%)	2.70 (96.1%)	1.24 (91.2%)
OML	channel flush	0.18 (0.8%)	0.16 (1.2%)	0.14 (1.8%)	0.03 (0.4%)	0.02 (0.4%)	0.09 (3.9%)	0.02 (0.3%)	0.02 (0.6%)	0.02 (1.0%)
	logging check	0.18 (0.8%)	0.16 (1.2%)	0.17 (2.2%)	0.41 (5.1%)	0.30 (6.2%)	0.19 (8.3%)	0.16 (2.9%)	0.16 (4.7%)	0.11 (5.4%)
	disk write	21.83 (87.8%)	13.11 (98.4%)	7.33 (97.6%)	7.6 (96.0%)	4.51 (94.5%)	2.01 (93.4%)	5.41 (96.8%)	3.22 (94.7%)	1.91 (93.6%)
CML	channel flush	0.18 (0.5%)	0.17 (0.7%)	0.12 (0.8%)	0.11 (0.5%)	0.12 (0.6%)	0.10 (1.0%)	0.01 (0.1%)	0.01 (0.2%)	0.01 (0.3%)
	disk write	35.60 (99.3%)	23.33 (99.1%)	14.66 (98.9%)	19.94 (99.2%)	18.35 (99.0%)	9.54 (98.5%)	9.15 (99.7%)	6.3 (99.5%)	3.72 (99.2%)
	garbage col.	0.06 (0.2%)	0.05 (0.2%)	0.04 (0.3%)	0.06 (0.3%)	0.07 (0.4%)	0.05 (0.5%)	0.02 (0.2%)	0.02 (0.3%)	0.02 (0.5%)

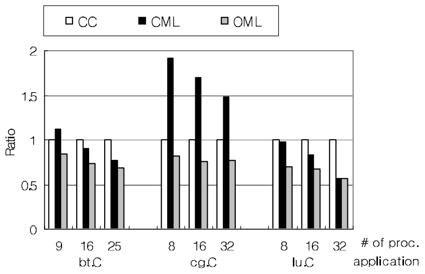


Fig. 5. The normalized re-execution time for the lost computation

5.3 Recovery Cost

To measure the recovery cost, we killed a certain process once at the middle between two consecutive checkpoints. The failed process recovered on the same node where it previously had run.

Figure 5 shows the re-execution time for the lost computation, that is to say, from the latest checkpointed state to the state right before it failed. We normalize each result to the normal execution time for the lost computation, therefore the value for coordinated checkpoint is naturally one. Message replaying usually makes re-execution faster than normal execution, since processes do not suffer from message delay. However, it is interesting that CML recovers the slowest in case of BT.C.9 and CG applications that transfer the huge amount of messages. It is due to the network congestion for re-sending large amount of message logs simultaneously. Although the amounts of logs for BT and CG are almost the same, BT recovers faster than CG, since the load of replaying is distributed further in BT; every process of BT has six neighbors, and a process of CG has two to four neighbors. In the other cases, CML and OML recover faster than CC protocol because replaying smaller amount of logs does not burden network load as much.

6 Conclusions

This paper presents the integration of consistent-recovery protocols into the industry-endorsed MPI programming model, and the performance comparison among those protocols. Few researches have compared coordinated checkpointing, causal message logging, and optimistic message logging altogether.

The results in this paper show that the coordinated checkpointing protocol outperforms in failure-free execution. Causal message logging is not suitable for communication intensive applications with large message amounts since it consumes memory exhaustively and it suffers from converging message replay on recovery time. Optimistic message logging suffers from disk access overhead even with asynchronous disk writes in our experiment, because disk buffer in kernel flushes frequently. Although it is superior in re-executing the lost computation and it is quite reliable, its excessive disk access is not favorable at all. Ironically, it makes the system unstable by overheating disk drives.

The main overhead in checkpointing/logging protocol is due to storing the checkpoint files into disks. There have been several efforts to reduce this overhead. Incremental checkpointing [13] stores only the changed portion of memory from the previous checkpoint, but it may not be so effective with the applications that calculate a huge matrix like NPB since the whole matrix changes every single iteration. Diskless checkpointing [14] transfers checkpoint files to the memory of neighbor nodes, but it is effectual only with small size of checkpoint files. Using application-level approach, developers can optimize the checkpoint content [12], but it does not seem to be a general solution. High degree of parallelization might help decreasing the checkpoint overhead since it divides and distributes the datasets, but the synchronization cost may increase.

In this paper, we let checkpoints be stored at the local disk. It is possible to transfer checkpoint/log files to remote stable storage server after local checkpointing. If each transfer is scheduled, we can avoid the congestion at the storage server.

Acknowledgements

The ICT at Seoul National University provides research facilities for this study.

The experimental results in this paper have been obtained on the cluster at Korea Astronomy Observatory. We thank its manager, Dr. Jongsoo Kim for his support.

References

1. L. Alvisi, E. N. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel : An Analysis of Communication Induced Checkpointing. FTCS-29, The 29th International Symposium on Fault-Tolerant Computing, pp. 242–249
2. L. Alvisi, and K. Marzullo : Trade-Offs in Implementing Causal Message Logging Protocols. *Proceedings of the 15th ACM Annual Symposium on the Principles of Distributed Computing*, pp.58–67, May 1996.
3. A. Agbaria and R. Friedman : Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Proc. IEEE Symp. on High Performance Distributed Computing*, pp. 167–176, August 1999.
4. A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello : Coordinated checkpoint versus message log for fault-tolerant MPI. *In proceedings of Cluster 2003*, pp. 242-250, December 2003.
5. K. M. Chandy and L. Lamport : Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computing Systems*, vol. 3, no. 1, pp. 63–75, Aug. 1985.
6. E. N. Elnozahy, L. Alvisi, Y.-M.Wang, and D. B. Johnson : A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.
7. W. Gropp, E. Lusk, N. Doss, and A. Skjellum : A high-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

8. N. T. Karnois, B. Toonen, and I. Foster : MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing.* vol. 63, no. 5, pp. 551–563, May 2003.
9. NASA Ames Research Center : Nas parallel benchmarks. *Technical report*, http://science.nas.nasa.gov/Software/NPB/, 1997.
10. N. Neves and W. K. Fuchs : RENEW: A tool for fast and efficient implementation of checkpoint protocols. *Symp. on Fault-Tolerant Computing*, pp. 58–67, 1998.
11. A. Nguyen-Tuong : Integrating Fault-Tolerance Techniques in Grid Applications. PhD thesis, University of Virginia, USA, 2000.
12. G. T. Nguyen, V. D. Tran, and M. Kotocová : Application recovery in parallel programming environment. *European PVM/MPI*, pp. 234–242, 2002.
13. J. S. Plank, M. Beck, G. Kingsley, and K. Li : Libckpt: Transparent checkpointing under unix. *USENIX Winter 1995 Technical Conference*, Jan. 1995.
14. J. S. Plank, K. Li, and M. A. Puening : Diskless checkpointing. *IEEE Trans. on Parallel and Distributed Systems*, vol. 9, no. 10, pp. 972–986, 1998.
15. S. Rao, L. Alvisi, and H. M. Vin : The cost of recovery in message logging protocols. *IEEE Transaction on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 160–173, Mar./Apr. 2000.
16. S. Rao, L. Alvisi, and H. M. Vin : Egida: An extensible toolkit for low-overhead fault-tolerance. *Symp. on Fault-Tolerant Computing*, pp. 48–55, 1999.
17. S. H. Russ, J. Robinson, B. K. Flachs, and B. Heckel : The Hector distributed run-time environment. *IEEE Trans. on Parallel and Distributed Systems*, vol. 9, no. 11, pp. 1102–1114, Nov. 1998.
18. G. Stellner : CoCheck: Checkpointing and process migration for MPI. *Proc. the Int'l Parallel Processing Symp.*, pp. 526–531, Apr. 1996.
19. V. Zandy : ckpt library. *http://www.cs.wisc.edu/ zandy/ckpt/.*
20. W. Zwaenepoel and E. N. Elnozahy : Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers*, Vol. C-41, No. 5, pp. 526-531, May 1992.

An Approach to Experimentally Obtain Service Dependability Characteristics of the Jgroup/ARM System

Bjarne E. Helvik[1], Hein Meling[2], and Alberto Montresor[3]

[1] Department of Telematics, Q2S Centre, Norwegian University of Science and Technology, O.S. Bragstadsplass 2A, N-7491 Trondheim, Norway
`bjarne@q2s.ntnu.no`

[2] Department of Electrical and Computer Engineering, University of Stavanger, N-4036 Stavanger, Norway
`meling@acm.org`

[3] Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy
`montresor@CS.UniBO.IT`

Abstract. Jgroup/ARM is a middleware framework for operating dependable distributed applications based on Java. Jgroup integrates the distributed object models of Java RMI and Jini with the *object group communication* paradigm, enabling the construction of groups of replicated server objects that provide dependable services to clients. ARM provides automated mechanisms for distributing replicas to host processors and recovering from replica failures.

This paper describes an approach based on *stratified sampling* combined with *fault injections* for estimating the dependability attributes of a service deployed using the Jgroup/ARM middleware framework. A first experimental evaluation is performed focusing on a service provided by a triplicated server, and indicative predictions of various dependability attributes of the service are obtained. The evaluation shows that a very high availability and MTBF may be achieved for services based on Jgroup/ARM.

Keywords: Fault Injection, Probabilistic Modeling and Evaluation, Measurement-based Evaluation, Failure Recovery, System Fault Tolerance, FT Middleware.

1 Introduction

The [illegible] paradigm for the development of dependable distributed applications has received considerable attention in recent years [19, 10, 20, 1]. Middleware frameworks based on this paradigm have been integrated with modern distributed object models like Java RMI and Jini [19, 6], and are currently being deployed in web-based business applications. Assessing and evaluating the dependability characteristics of such frameworks, however, have not received an equal amount of attention.

To fill this void, this paper presents an extensive evaluation of Jgroup/ARM [19, 5, 17], one such framework. Jgroup enables the construction of dependable applications based on [illegible] of replicated server objects, that cooperate in order to provide dependable services to their clients. Communication inside a group, as well as communication between clients and server objects, is based on [illegible] that are executed by all members of the group. ARM builds on Jgroup, providing mechanisms for the automated management of groups, by distributing replicas to host processors and recovering from replica failures.

In the evaluation, dependability attributes are predicted through a [illegible] [14] approach. A series of experiments are performed; in each of them, one or more faults are injected according to an accelerated homogeneous Poisson process. The approach defines strata in terms of the number of near-coincident failure events that occur in a fault injection experiment. By near-coincident is meant failures occurring before the previous is handled. Hence, [illegible] is performed where experiments are allocated to strata after they are carried out. This as opposed to the more common prior stratification where strata are defined before the experiment. Three strata are considered, i.e., single failures, and double and triple near-coincident failures. The system under study is assumed to follow a crash failure semantics. For the duration of an experiment, the events of interest are monitored, and post-experiment analysis is performed to construct a single global timeline of fault injections and other relevant events. The timeline is used to compute trajectories on a predefined state machine.

Depending on the number of injected faults, each experiment is classified into one of the strata, and various statistics for the experiments are obtained. These statistical measures are then used as input to an estimator of dependability attributes, including [illegible], [illegible] and [illegible]. The approach may also be used to find periods with reduced performance due to fault handling. An additional benefit of this thorough evaluation is that the fault handling capability of Jgroup/ARM has been tested extensively, enabling the discovery of rarely occurring implementation faults of both the distributed service under study and the Jgroup/ARM framework itself.

Fault injection is a valuable and widely used means for the assessment of fault tolerant systems, see for instance [2, 3, 11]. Previously, stratified sampling has been used in combination with fault injection experiments to estimate fault tolerance coverage, as presented in [9]. Furthermore, for testing specific parts of a system, fault injection triggers has been used on a subset of the global state space [8]. These approaches are very useful in testing and evaluating specific aspects of a system. However, our objective is to perform an overall evaluation of the system and its ability to handle processor failures and hence, random injections of crash failures in a operational system and post stratification is applied.

Delta-4 provide fault treatment mechanisms similar to those of ARM [21]. Fault injections were also used in Delta-4 [4], focusing on removal of design/implementation faults in fault tolerance mechanisms. However, we are not aware of reports on the evaluation of the fault treatment mechanisms in Delta-4, comparable to those presented herein. The fault injection scheme used in this work, combined with post-experiment analysis also facilitate detection of implementation faults, and in addition allows for systematic regression testing.

The AQuA [23, 22] framework is based on CORBA and also support failure recovery. Unlike Jgroup/ARM, it does not deal with partition failures and relies on the open group model [13] which limits its scalability with respect to supporting a large number of groups. The evaluation of AQuA presented in [22] only provide the various delays involved in the recovery time. In this paper, focus is on estimating dependability attributes of services deployed through ARM.

[illegible] Section 2 provides an overview of the features of the Jgroup/ARM framework relevant to this paper. Section 3 describes the target system for our measurements, while Section 4 presents the measurement setup and strategy, together with the associated estimators. Experimental results are given in Section 5, and Section 6 concludes the paper.

2 The Jgroup/ARM Middleware

2.1 Jgroup

Jgroup [5] integrates the Java RMI and Jini [illegible] with the [illegible] paradigm and includes numerous innovative features that make it suitable for developing modern network applications. In Jgroup, applications are based on collections of replicated [illegible] that cooperate to provide a dependable service. For increased flexibility, the group composition is allowed to vary dynamically as new servers [illegible] and existing servers [illegible] the group, either voluntarily or by crashing. Members of the group are kept informed about the current group composition through a [illegible] (GMS).

Communication facilities for the object group are provided by the [illegible] (GMIS), that enables the execution of remote method invocations on all members of an object group. Jgroup is unique in providing such uniform object-oriented programming interface to govern [illegible] object interactions, including those within an object group as well as interactions with external objects (clients). Both the GMS and the GMIS have been formally specified, admitting formal reasoning about the correctness of applications based on these services [5, 19]. Due to space constraints, however, in the following only a short informal description is provided, focusing on the properties that are needed to understand this paper.

At any time, the [illegible] of a group includes those servers that are operational and have joined, but have not yet left the group. Asynchrony of the system and failures may cause each member to have a different perception of the group's current membership. The task of the GMS is to notify members about variation in the group membership. These notification are called [illegible]; we say that a node [illegible] a view when such notification is delivered to it. A view consists of a list of nodes along with a unique identifier, and corresponds to the group's current composition as perceived by members included in the view.

View changes must satisfy the following requirements [5]. First, the service must track changes in the group membership accurately and in a timely manner such that installed views indeed convey recent information about the group's composition. Next, a view can be installed only after agreement is reached on its composition among the servers included in the view. Finally, GMS must guarantee that two views installed by two different servers be installed in the same order. Note that the GMS defined for Jgroup admits coexistence of concurrent views, each corresponding to a different partition of the communication network, thus making it suitable for partition-aware applications.

Group method invocations must satisfy a variant of [illegible], that has proven to be an important property for reasoning about reliability in message-based systems [7]. Informally, view synchrony requires that two servers that install the same pair of consecutive views agree to complete the same set of invocations during the first view of the pair. In other words, before a new view can be installed, all servers belonging to both the current and the new view have to agree on the set of invocations they have completed in the current view.

The agreement properties of Jgroup (on view composition and invocation execution) enable a server to reason about the state of other servers in the group using only local information such the history of installed views and the set of completed group method invocations.

2.2 Autonomous Replication Management (ARM)

Most existing object group systems do not include mechanisms for distributing replicas to host processors or recovering from replica failures. Yet, these mechanisms are essential for satisfying application dependability requirements such as maintaining a fixed redundancy level. The ARM framework provides a replicated dependability manager that enables the autonomic management of complex applications based on object groups [15, 17]. When installed, an object group becomes an "autonomous" entity being maintained by ARM, until it is explicitly removed. During its life, an object group provides service to clients completely decoupled from the ARM infrastructure.

ARM handles both replica distribution, according to an extensible [illegible], as well as replica recovery, based on a [illegible]. The replication policy is group-specific, and allows the creation of object groups with varying dependability requirements and recovery needs. The distribution policy is specific to each ARM deployment, and requires configuration of the set of processors on which replicas can be created.

The ARM framework consists of several components: a system-wide [illegible] (RM), [illegible] deployed at each of the managed replicas, and [illegible] deployed at each of the processors. Recovery modules are responsible of forwarding view change notifications to the RM, that will interpret this information, potentially triggering group-specific actions like replica creation or removal.

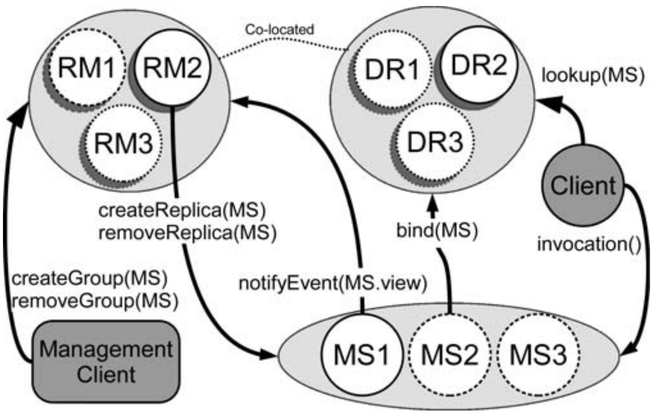


Fig. 1. The Jgroup/ARM architecture

Together, these components form a [illegible] facility, whose goal is to re-establish desired system dependability properties after failures.

Fig. 1 gives a simplified overview of a typical ARM-based deployment, and associated communication patterns. The system contains a single replicated service, named MS, that is managed by ARM. In each of the groups, a [illegible] replica is elected. Every view change event generated by the group communication system is reported by the recovery module of the MS leader to the RM using notifyEvent(). The RM interprets the received event, and the leader issues a createReplica() or removeReplica() call to the object factories of a selected set of processors, depending on the distribution policy. The RM provides an external interface, composed of method createGroup() and removeGroup(), that enables the management client to start or stop services. Fig. 1 also illustrates how servers bind their reference in the dependable registry (DR) service, and how clients query this naming service to obtain information required to communicate with the server group through method invocations. Note that replicas of the RM and DR groups are co-located on the same set of nodes. For simplicity, the illustration of the target system (see Fig. 2) depict only the RM group.

In addition to the above, there is also a mechanism embedded in the recovery module for recovering from scenarios in which the whole application group has failed. This is accomplished using a lease renewal technique, requiring that the leader of each deployed group issues a renew event periodically to prevent the RM group from triggering recovery.

The RM group provides self-recovery by reusing the same mechanisms that are in place to track other applications, except for the lease renewal. Another important feature of ARM is its ability to handle multiple concurrent failures, even failures of the ARM framework itself, as long as at least one RM replica remains.

3 Target System

Fig. 2 shows the target system for our measurements. It consists of a cluster with a total of $n = 8$ identical processors, initially hosting a single server replica as shown. In the experiments, ARM uses a distribution policy that will avoid co-locating two replicas of the same type, and at the same time it will try to keep the replica count per processor to a minimum. Different services may share the same processor.

The ARM infrastructure (i.e., the RM group) is located on processors (1-3). Processors 5-7 host the [illegible] (MS), while processors 4 and 8 host the [illegible] (AS). The latter was added to assess ARM's ability to handle multiple concurrent failure recoveries at different groups, and to provide a more realistic scenario. Finally, an external machine hosts the [illegible] that is used to run the experiments; for a description of the experiment engine, please refer to Section 4.2. The replication policy for all the deployed services requires that ARM tries to maintain a fixed redundancy level (RM:=3, MS:=3, AS:=2), with the RM group being at least as fault-tolerant as the remaining components of the system.

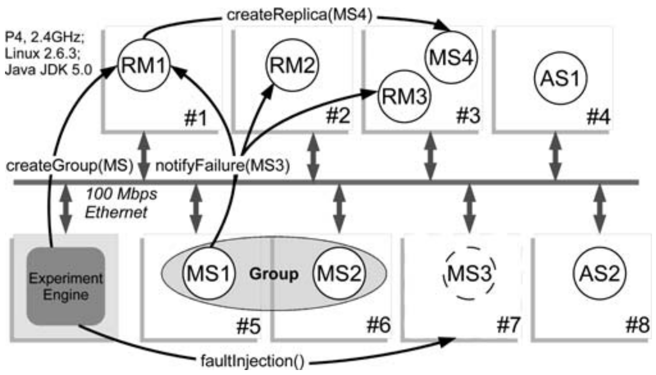


Fig. 2. Target system illustrated

The measurement engine enables the simultaneous observations of all services in the target system, including the ARM infrastructure. In the following, however, we will focus our attention on the MS service, that constitutes our [illegible]. This subsystem will be the subject of our observations and measurements, with the aim of predicting its dependability attributes. Note that focusing on a particular subsystem of interest is for simplifying presentation. Observations of several subsystems could be done simultaneously and estimates/predictions of all services and the ARM infrastructure may be obtained during the same experiment.

3.1 The State Machine

There is a set of states which we can observe and which are sufficient to determine the dependability characteristics of the service(s) regarded. Note that these are not necessarily all the operational states of the complete system, but the set of states associated with the MS service. Thus, the failure-recovery behavior of the MS service can be modeled according to the [illegible] in Fig. 3, irrespective of the states of the ARM and AS subsystems. The state diagram is not used to control fault injections based on triggers on a subset of the global state space as in [8]; instead it is only used during offline, a posteriori analysis of fault injection experiments based on random sampling. In the analysis, the trajectories on the state model, and the time spent in each state is used together with mathematical tools to determine the dependability characteristics of the MS service.

We define a service to be [illegible] (squared states) if none of the group members have installed a view, and [illegible] (circular states) if at least one member has installed a view. Each state is identified by $X\sharp$ and a tuple $(x\mathbf{r}, y\mathbf{v})$, were x is the number of installed replicas ($\mathbf{r}$), and y is the number of members in the current view ($\mathbf{v}$) of the server group. In the diagram, we consider only events that may affect the availability of the service, such as [illegible], [illegible] as seen from the perspective of ARM, and [illegible] as perceived by the corresponding MS nodes that fails. View changes, in particular, are denoted by [illegible], where i is the cardinality of the view. In addition, [illegible] events may occur in any of the states, however for readability they are not included in the figure.

As a sample failure-recovery behavior, consider the trajectory composed of the state transitions with dashed lines, starting and ending in $X0$. This is the most common trajectory. For simplicity the [illegible] events in the diagram reflect the series of views as seen by ARM, and do not consider the existence of concurrent views. So, after recovering from a failure (moving from state $X4$ to state $X3$), the newly created member will install a singleton view and thus be the leader of that view, sending a [illegible] event to ARM (from state $X3$ to state $X6$). Only after this installation (required by the view synchrony property) a [illegible] event will be delivered to ARM, causing a transition from state $X6$ to $X0$. The above simplification does not affect the availability of the service. It is assumed that client requests are only delayed during failure-recovery cycles as long as the service is in an operational state [16]. Such delays are not considered part of the availability measure as opposed to [12]. Further analysis of these client perceived delays is in preparation and will be included in a future paper.

Notice that some of the states have self-referring transitions on view change events. These are needed for several reasons, one being that the ARM framework may see view change notifications from several replicas, before they have formed a common view. In addition, ARM will on rare occasions receive what we call [illegible], that are due to minor inaccuracies in our measurements. For instance, a [illegible] event may occur while in state $X7$. This can occur if at

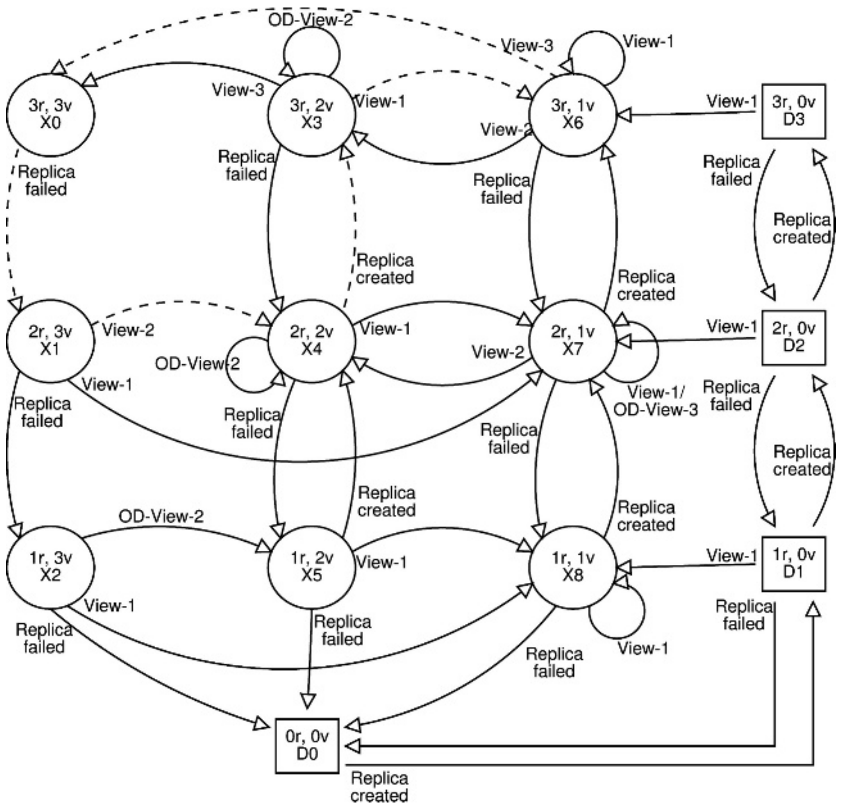


Fig. 3. State diagram illustrating a sample of the possible state changes of the MS service being measured

some point we are in the $X6$ state, when a group member sends out a notification of a [illegible] event, and shortly after another member of that group fails and logs a *ReplicaFailed* event. However, given that the [illegible] event is still in the "air", and has not yet been logged, the *ReplicaFailed* event will appear to have occurred before the [illegible] event in the trace. To compensate for this behavior, we have inserted additional [illegible] transitions, prefixed by OD, in some of the states.

Notice also the [illegible] transition from $X2$ to $X5$. This is also due to an outdated view, and can occur if ARM triggers recovery on a [illegible] event before receiving a [illegible] event. Note that the state transitions in the diagram may not be complete as presented, however, no other transitions have been observed during our experiments. In the following, we will assume that the service has been initialized correctly into state $X0$, and thus we do not consider the initial transitions leading to this state.

4 Measurements

This section give motivation for our measurement approach. Furthermore, we discuss in detail the sampling scheme used to assess the fault handling capability of the Jgroup/ARM framework and to provide input to the prediction of dependability attributes.

4.1 Experiment Outline

In each experiment run, one or more faults are injected. The failure insertion pattern is as if it emerged from a Poisson process. There may be multiple near-coincident failures before the system stabilizes, i.e., a new failure may be inserted before the previous has been completely handled. This will "simulate" the rare occurrence of nearly coincident failures which may bring the service down. The Poissonian character of the inserted failures is achieved through generation of fault injection times and the selection of the set of processors in which to inject faults, according to a uniform distribution. See the Sampling Scheme in Section 4.3 on how this yields a Poisson fault process. Processors to crash are drawn from the entire target system. Hence, the injected faults may affect the ARM infrastructure itself, the monitored subsystem (MS) or the additional service (AS), all of which are being managed by the ARM framework. However, only state trajectories for the monitored subsystem are computed, and these are used for predicting various dependability attributes of MS. A beneficial "side-affect" of this sampling scheme is that it has shown to be very useful with respect to performing extensive testing of the fault handling capabilities of the Jgroup/ARM. During previous experiments several design and implementation faults have been revealed. In the experiments, we perform at most $k = 3$ fault injections during a run. Since initially all processors in the target system have allocated replicas, failures will cause ARM to reuse processors as shown in Fig. 2 where the replica of processor 7 is recreated at processor 3.

Time Constants Considered. Assuming services are deployed using the ARM framework, the crashed processors will have a [illegible] (t_{PR}) which is much longer than the [illegible] (t_{SR}). Further, we assume that the processors will stay crashed for the remaining part of the experiment. In other words, a service replica will typically be restarted on a different processor as soon as ARM concludes that a processor crash has occurred. However, the time until the processors are recovered, is assumed to be negligible compared to the [illegible] (t_{BF}) in a real system. Thus in the predictions it is assumed that the occurrence intensity of new trajectories (i.e. first failure in a fully recovered system) is $n\lambda$, neglecting the short interval with a reduced number of processors between t_{SR} and t_{PR}. Fig. 4 shows these relations, starting with the first failure event t_{i_1}. Furthermore, there will be no resource exhaustion, i.e., there are sufficient processors to execute all deployed services, including the ARM infrastructure.

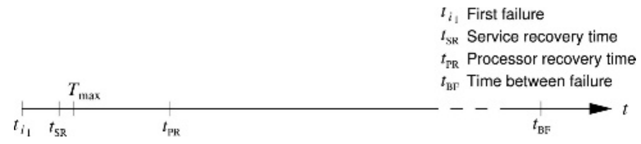

t_{i_1} First failure
t_{SR} Service recovery time
t_{PR} Processor recovery time
t_{BF} Time between failure
T_{max}
t
t_{i_1}
t_{SR}
t_{PR}
t_{BF}

Fig. 4. The relation between the service and processor recovery periods and the time between failures

The Failure Trajectory. A failure trajectory is the series of events and states of the monitored subsystem following the first processor failure and until all the concurrent failure activities have concluded and all subsystems are recovered and fully replicated. The trajectory will always start and end in state $X0$ (see Fig. 3). the first processor failure affects the monitored service, it causes it to leave its steady operational state $X0$ and , it is the last service to recover, we will see a return to the same state like in Fig. 5.

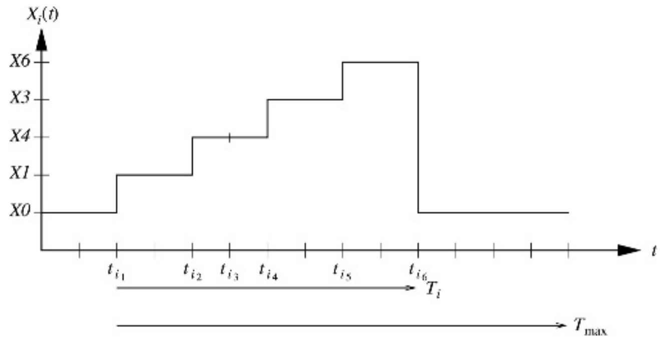


Fig. 5. Sample failure trajectory, where all but failure event t_{i_3} affects the subsystem of interest. This is the most common failure trajectory

We denote the j^{th} event in the i^{th} trajectory by i_j, the time it takes place by t_{i_j} and the state after the event by X_{i_j} (corresponds to the states in Fig. 3). Note that all relevant events in the system are included, and a failure or another event does not necessarily cause a change of state in the monitored subsystem. For instance, the failure of a processor which supports only the ARM or AS subsystems, will not necessarily result in a change of state in the MS service, but it is likely that it will influence the handling of immediately preceding or succeeding failures affecting the service. Let $X_i(t)$ denote the state of the MS service at time t in the i^{th} failure trajectory,

$$X_i(t) = \begin{cases} X_{i_j} & t_{i_j} < t \leq t_{i_{j+1}},\ j = 1, \ldots, m_i \\ X0 & \text{Otherwise} \end{cases}$$

were m_i is the last event of the i^{th} trajectory before all concurrent failure activities have concluded, and all subsystems are fully replicated. During the measurements a trajectory sample is recorded as the list

$$\underline{X}_i = \left\{X0, t_{i_1}, X_{i_1}, t_{i_2}, X_{i_2}, t_{i_3}, \ldots, t_{i_{m_i}}, X0\right\}.$$

Note that we record also trajectories for which the MS service does not leave the $X0$ state.

Characteristics Obtained from a Failure Trajectory. The unknown probability of failure trajectory i is p_i. For brevity we denote the duration of trajectory i by $T_i = t_{i_{m_i}} - t_{i_1}$, and its expectation $\Theta = E(T) = \sum_{\forall i} p_i T_i$.

In the following, let Y_i denote a sample from the experiment. The sample may be obtained from the trajectory by some function g, i.e., $Y_i = g(\underline{X}_i)$. The duration of a trajectory presented above may serve as an example. Determining the dependability attributes of the system are other possible samples that can be extracted from the experiment data. To determine these, it is assumed that the failure rate in the $X0$ state is $n\lambda$, that the expected sojourn time in this state is much longer than the expected trajectory duration, and that a particular trajectory is independent of the previous trajectory.

[illegible] The time spent in a down state during a trajectory is given by

$$Y_i^d = g(\underline{X}_i) = \sum_{j=1}^{m_i} I(X_{i_j} \in \mathfrak{F})(t_{i_{j+1}} - t_{i_j}), \tag{1}$$

where $I(\cdots)$ is the indicator function and $\mathfrak{F}$ is the set of down states (the squared states in Fig. 3). Given that the periods in state $X0$ (the OK-periods) alternate with the failure trajectories, and are independent and much longer than the failure trajectory periods, we can obtain a measure for the service unavailability

$$\hat{U} = \frac{E(Y^d)}{E(Y^d) + (n\lambda)^{-1}} \approx E(Y^d) n\lambda.$$

Note that the collective failure intensity of all processors when there are no faults in the system, is only marginally different from the intensity of trajectories. The difference is due to the restoration of failed processors during a trajectory, and is negligible.

[illegible] In this case, let $Y_i^f = 1$ if trajectory i visits one or more down states, otherwise let $Y_i^f = 0$.

$$Y_i^f = g(\underline{X}_i) = I(\exists X_{i_j} \in \mathfrak{F})_{j=1,\ldots,m_i}. \tag{2}$$

Disregarding multiple down periods in the same trajectory and assuming that system failures are rare, it is found that the system failure intensity is approximately

$$\hat{\Lambda} = \frac{1}{\text{MTBF}} \approx \frac{E(Y^f)}{E(Y^d) + (n\lambda)^{-1}} \approx E(Y^f) n\lambda.$$

In addition, the predicted reliability function $R(t) = \exp(-\hat{\Lambda} t)$ as well as the mean down time $\text{MDT} = \hat{U}/\hat{\Lambda}$ may be obtained. MDT and the down time distribution may of course also be measured directly from the trajectories visiting the set of down states.

The above examples are chosen for illustration and the assumptions made for simplicity. By introducing rewards associated with the states and transitions, we may obtain predictions of far more comprehensive performability measures of the system.

4.2 The Experiment Engine

The [illegible] is used to perform repeated experiment runs. At each run, numerous tasks are executed; (1) bootstrap the object factories onto the processors in the target system, (2) bootstrap the ARM infrastructure, (3) deploy the initial MS and AS replicas, (4) perform fault injections, according to the scheme described in Section 4.3, (5) shutdown the experiment run, and finally, (6) collect and remove log files from the target system.

To be able to compute a trajectory of states and the time spent in each state, Jgroup/ARM and the MS service has been instrumented with a simple event logging mechanism to be able to generate a local trace of events occurring at each of the processors in the target system. The events in a trace correspond to the events of the state diagram in Fig. 3. Each event trace contains the set of events and their occurrence time (in milliseconds), in addition to various details associated with the event. This level of accuracy is sufficient for our evaluation, as the time values considered (t_{SR}) are in the range 7-30 seconds. The occurrence time of an event correspond to the local clock of the processor at which the event occurred. The processor clocks in the target system are synchronized using NTP [18].

After each experiment, the log files generated are collected from the target system and stored at the experiment engine machine, for the offline analysis. In this analysis, the independent event traces collected from different machines are merged into a single global timeline of events, that correspond to an approximation of the actual state transitions of the whole system. Given this global event trace, we can compute the trajectory of visited states and the time spent in each of the states. These trajectories allow us to classify the experiments, and to predict a number of dependability attributes for the monitored service, as discussed previously.

4.3 Experimental Strategy

The experimental strategy is based on a [illegible] approach. For an introduction to stratified sampling see for instance [14]. This section elaborates on how the experiments are classified in different strata, and how the sampling is performed.

Stratification. Only some of the events along a failure trajectory will actually be failure events. The first event of each trajectory will always be a failure, and in a typical operational environment usually the only one. However, in the experiments we consider also multiple near-coincident failures which may require concurrent failure handling. In considering such failure scenarios, our experimental strategy is based upon subdividing the trajectories into strata S_k based on

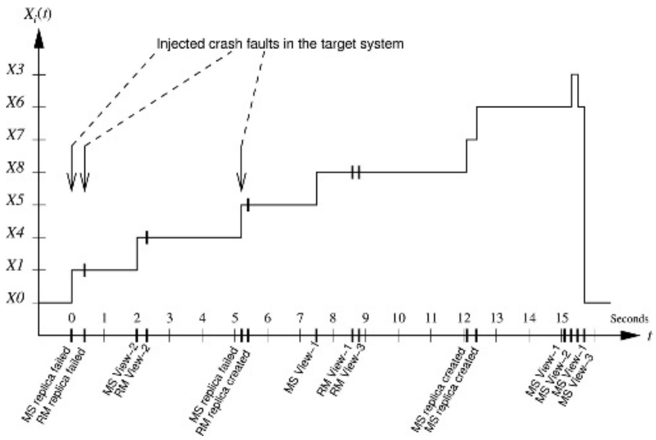


Fig. 6. Sample failure trajectory reaching stratum S_3 plotted on an approximate time scale. Only two of three injected faults affect the MS service. The second fault injection affect the RM service

the number of failure events k in each of the trajectories. Each of the strata are sampled separately, and the number of samples in each stratum are random variables determined a posteriori. This is different from previous work [9] in which the number of samples in each stratum is fixed in advance.

An example failure trajectory reaching stratum S_3 drawn from the experiment data is shown in Fig. 6. Three near-coincident fault injections were performed in this particular experiment. The first and last failure affect the MS service, while the second affect the RM service. The RM failure and its related events, as indicated on the curve, do not cause state transitions in the state diagram (Fig. 3) of the MS service.

The collected samples for each stratum are used to obtain statistics for the system in that stratum, e.g., the expectation $E(Y|S_k)$. The expectation and the variance of the length of the trajectory within a stratum S_k are denoted $\Theta_k = E(T|S_k)$ and $\sigma_k = Var(T|S_k)$, respectively. Estimates may then be obtained by

$$E(Y) = \sum_{k=1}^{\infty} E(Y|S_k)\pi_k \approx \sum_{k=1}^{3} E(Y|S_k)\pi_k, \tag{3}$$

where $\pi_k = \sum_{\forall i \in S_k} p_i$ is the probability of a trajectory in stratum S_k. Recall that k represents the number of possible concurrent failure events, and in (3), we replace ∞ in the summation with 3, since we only consider up to 3 concurrent failure events. Expressions for π_k are derived in Section 4.4.

If upper and lower bounds for Y exist, and we are able to determine $\pi_k, k > 3$, we may also determine bounds for $E(Y)$ without sampling the higher-order strata, i.e.,

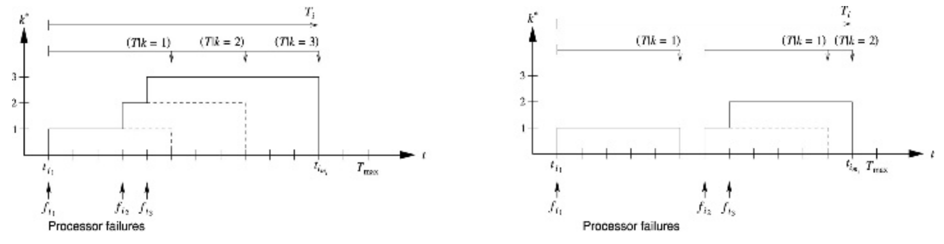
$$\sum_{k=1}^{3} E(Y|S_k)\pi_k + \inf(Y)\sum_{k>3}\pi_k \leq E(Y) \leq \sum_{k=1}^{3} E(Y|S_k)\pi_k + \sup(Y)\sum_{k>3}\pi_k.$$

Since the probability of k concurrent failures is much greater than $k+1$ failures, $\pi_k \gg \pi_{k+1}$, the bounds will be tight, and for the estimated quantities the effect of estimation errors are expected to be far larger than these bounds. The effect of estimation errors is discussed in Section 4.4.

Sampling Scheme. Under the assumption of a homogeneous Poisson fault process with intensity λ per processor, it is known that if we have $k-1$ faults (after the first failure starting a trajectory) of n processors during a fixed interval $[0, T_{\max}\rangle$, these will occur

- uniformly distributed over the set of processors, and
- each of the faults will occur uniformly over the interval $[0, T_{\max}\rangle$.

Note that, all injected faults will manifest itself as a failure, and thus the two terms are used interchangeably. In performing experiments, the value $T_{\max}$ is chosen to be longer than any foreseen trajectory of stratum S_k. However, it should not be chosen excessively long, since this may result in too rare observations of higher-order strata.



(a) Fault injections causing the failure trajectory into higher-order strata.

(b) Failure trajectory that completes before reaching higher-order strata.

Fig. 7. Sample failure trajectories with different fault injection times

In the following, let $(T|k=l)$ denote the duration of a trajectory if it completes in stratum S_l, as illustrated in Fig. 7(a), and let f_{i_l} denote time of the l^{th} failure, relative to the first failure event in the i^{th} failure trajectory. That is, we assume the first failure occur at $f_{i_1} = 0$ and that $f_{i_l} > f_{i_1}, l > 1$. To obtain dependability characteristics for the system, we inject k failures over the interval $[0, T_{\max}\rangle$. This leads to the following failure injection scheme for trajectory i, which will reach stratum S_k. However, not all trajectories obtained for experiments with $k > 1$ failure injections will reach stratum S_k, since a trajectory may reach $(T|k=1)$ before the second failure (f_{i_2}) is injected. That is, recovery from the first failure may complete before the second failure injection, as illustrated in Fig. 7(b). Such experiments contain multiple trajectories, however, only the first trajectory is considered in the analysis to avoid introducing a bias in the results.

1. The first failure, starting a failure trajectory i, is at $f_{i_1} = 0$. The following $k-1$ failure instants are drawn uniformly distributed over the interval $[0, T_{\max}\rangle$ and sorted such that $f_{i_q} \leq f_{i_{q+1}}$ yielding the set $\{f_{i_1}, f_{i_2}, \ldots, f_{i_k}\}$. Let k^* denote the reached stratum, and l is the index denoting the number of failures injected so far. Initially, set $k^* := 0$ and $l := 1$.
2. Fault $l \leq k$ is (tentatively) injected at f_{i_l} in processor $z_l \in [1, n]$ with probability $1/n$.
   (a) If trajectory i has not yet completed, i.e., $f_{i_l} < T_i$, then set $l := l + 1$ and
      i. If the selected processor has not already failed $z_l \notin \{z_w | w < l\}$: [illegible] at f_{i_l} and set $k^* := k^* + 1$
      ii. Prepare for next fault injection, i.e., goto 2.
   (b) Otherwise the experiment ended "prematurely".
3. Conclude and classify as a stratum S_{k^*} measurement.

The already failed processors are kept in the set to maintain the time and space uniformity corresponding to the constant rate Poisson process. Although k failures are not injected in a trajectory, the pattern of injected failures will be as if they came from a Poisson process with a specific number (k^*) of failures during $T_{\max}$. Hence, the failure injections will be representative for a trajectory lasting only a fraction of this time.

4.4 Estimators

Strata Probabilities. In a real system, the failure intensity λ will be very low, i.e., $\lambda^{-1} \gg T_{\max}$. Hence, we may assume the probability of a failure occurring while the system is on trajectory $i \in S_1$ is $T_i(n-1)\lambda$. Hence, the probability that a trajectory (sample) belonging to a stratum $S_k, k > 1$ occurs, given that a stratum S_1 cycle has started is

$$\frac{\sum_{\forall i \in S_1} p_i T_i (n-1)\lambda}{\sum_{\forall i \in S_1} p_i} = \frac{\sum_{k>1} \pi_k}{\pi_1}.$$

Due to the small failure intensity, we have that $\sum_{k>1} \pi_k \approx \pi_2$ and the unconditional probability of a sample in stratum S_2 is approximately

$$\pi_2 = (n-1)\lambda \Theta_1 \pi_1. \tag{4}$$

This line of argument also applies for the probability of trajectories in stratum S_3. However, in this case we must take into account the first failure occurrence. Let $i \in S_k \wedge X_i(t_x) \bowtie \mathfrak{f}$ denote a trajectory of stratum S_k, where a failure occurs at t_x. The probability that a trajectory belonging to stratum $S_k, k > 2$ occurs, given that a stratum S_2 cycle has started is, cf. Fig. 7(a):

$$\frac{\int \sum_{\forall i \in S_2 \wedge X_i(t_x) \bowtie \mathfrak{f}} p_i (T_i - t_x)(n-2)\lambda dt_x}{\sum_{\forall i \in S_2} p_i} = \frac{\sum_{k>2} \pi_k}{\pi_2}. \tag{5}$$

Ignoring the constant part of (5) for now; the first term on the left hand side of (5) is not depending on t_x and may be reduced as follows:

$$\frac{\int \sum_{\forall i \in S_2 \wedge X_i(t_x) \bowtie \mathfrak{f}} p_i T_i dt_x}{\sum_{\forall i \in S_2} p_i} = \frac{\sum_{\forall i \in S_2} p_i T_i}{\sum_{\forall i \in S_2} p_i} = \Theta_2 .$$

For the second term we have, slightly rearranged:

$$\int t_x \sum_{\forall i \in S_2 \wedge X_i(t_x) \bowtie \mathfrak{f}} p_i dt_x .$$

The probability of having a stratum S_2 trajectory experiencing its third failure at t_x is the probability that the first (and second) failure has not been dealt with by t_x, i.e., the duration $T_j > t_x, j \in S_1$ and that a new failure occurs at t_x. These two events are independent. Up to the failure time t_x, the trajectories of strata S_1 and S_2 passing this point are identical. Hence, $\sum_{\forall i \in S_2 \wedge X_i(t_x) \bowtie \mathfrak{f}} p_i = \Pr\{T_j > t_x\}\pi_1(n-1)\lambda$ and by partial integration,

$$\int t_x \Pr\{T_j > t_x\} dt_x = \frac{1}{2} E(T_j^2 | j \in S_1) = \frac{1}{2}(\Theta_1^2 + \sigma_1).$$

Combining the above, inserting it into (5), using that $\sum_{\forall i \in S_2} p_i = \pi_2$ and that due to the small failure intensity $\sum_{k>2} \pi_k \approx \pi_3$, the unconditional probability of a trajectory in stratum S_3 approximately becomes:

$$\begin{aligned} \pi_3 &= (n-2)\lambda(\Theta_2 \pi_2 - \frac{1}{2}(\Theta_1^2 + \sigma_1)\pi_1(n-1)\lambda) \\ &= (n-1)(n-2)\lambda^2(\Theta_2\Theta_1 - \frac{1}{2}(\Theta_1^2 + \sigma_1))\pi_1 . \end{aligned} \tag{6}$$

Since we have that $1 > \pi_1 > 1 - \pi_2 - \pi_3$ and as argued above, a sufficiently accurate estimate for π_1 may be obtained from the lower bound since $1 \approx \pi_1 \approx 1 - \pi_2 - \pi_3$, or slightly more accurately by solving π_i from (4), (6) and $1 = \pi_1 + \pi_2 + \pi_3$.

Estimation Errors. The estimation errors or the uncertainty in the obtained result is computed using the sectioning approach [14]. The experiments are subdivided into $N \sim 10$ independent runs of the same size. Let $\hat{E}_l(Y)$ be the estimate from the l^{th} of these; then:

$$\hat{E}(Y) = \frac{1}{N}\sum_{l=1}^{N} \hat{E}_l(Y), \quad \hat{\text{Var}}(Y) = \frac{1}{(N-1)}\sum_{l=1}^{N}(\hat{E}_l(Y^2) - \hat{E}^2(Y)).$$

5 Experimental Results

This section presents experimental results of fault injections on the target system. A total of 3000 experiment runs were performed, aiming at 1000 per stratum. Each experiment is classified as being of stratum S_k, if exactly k fault injections occur before the experiment completes (all services are fully recovered).

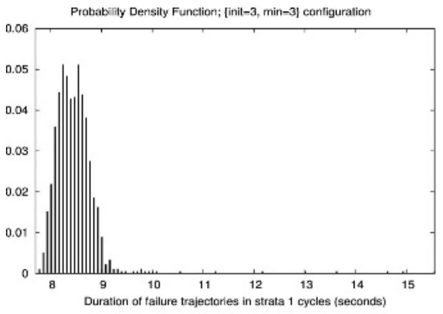
The results of the experiments are presented in Table 1. Some runs "trying to achieve higher order strata" (S_3 and S_2) fall into lower order due to injections being far apart, cf. Fig. 7(b), or addressing the same processor.

Table 1. Results obtained from the experiments (in milliseconds)

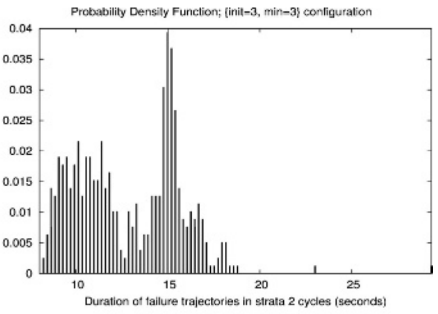
Classification	Count	$\Theta_k = E(T \mid S_k)$	sd=$\sqrt{\sigma_k}$	Θ_k, 95% conf.int.
Stratum S_1	1781	8461.77	185.64	(8328.98, 8594.56)
Stratum S_2	793	12783.91	1002.22	(12067.01, 13500.80)
Stratum S_3	407	17396.55	924.90	(16734.96, 18058.13)

Of the 3000 runs performed, 19 (0.63%) were classified as inadequate. In these runs one or more of the services failed to recover (16 runs), or they behaved in an otherwise unintended manner. In the latter three runs, the services did actually recover successfully, but the runs were classified as inadequate, because an additional (not intended) failure occurred. The inadequate runs are dispersed with respect to experiments seeking to obtain the various strata as follows; two for S_1, 6 for S_2, and 11 for stratum S_3. One experiment resulted in a complete failure of the ARM infrastructure, caused by three fault injections occurring within 4.2 seconds leaving no time for ARM to perform self-recovery. Of the remaining, 13 were due to problems with synchronizing the states between the RM replicas, and 2 were due to problems with the Jgroup membership service. Even though none of the inadequate runs reached the down state, $D0$, for the MS service, it is likely that additional failures would have caused a transition to $D0$. To be conservative in the predictions below, all the inadequate runs are considered to have trajectories visiting down states, and causing a fixed down time of 5 minutes.

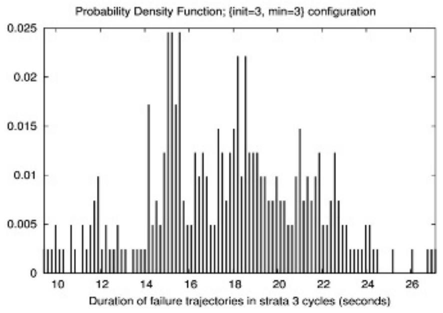
Fig. 8 shows the probability density function (pdf) of the recovery periods for each of the strata. The data for stratum S_1 cycles indicate that it has a small variance. However, 7 runs have a duration above 10 seconds. These durations are likely due to external influence (CPU/IO starvation) on the machines in the target system. This was confirmed by examining the cron job scheduling times, and the running time of those particular runs. Similar observations can be identified in stratum S_2 cycles, while it is difficult to identify such observations in S_3 cycles. The pdf for stratum S_2 in Fig. 8(b) is bimodal, with a top at approx. 10 and another at approx. 15. The density of the left-most part is due to runs with injections that are close, while the right-most part is due to injections that are more than 5-6 seconds apart. The behavior causing this bimodality is due to the combined effect of the delay induced by the view agreement protocol, and a 3 second delay before ARM triggers recovery. Those injections that are close tend to be recovered almost simultaneously. The pdf for stratum S_3 has indications of being multimodal. However, the distinctions are not as clear in this case.



(a) Stratum S_1



(b) Stratum S_2



(c) Stratum S_3

Fig. 8. Probability density function of trajectory durations for the various strata

Given the results of the experiments, we are able to compute the expected trajectory durations, Θ_1, Θ_2 and the variance σ_1 as shown in Table 1, and the unconditional probabilities π_2 and π_3 given in (4) and (6) for various processor mean time between failures (MTBF=λ^{-1}), as shown in Table 2. The low probabilities of a second and third near-coincident failure is due to the relatively short recovery time (trajectory durations) for strata S_1 and S_2. Table 2 compares these values with a typical processor recovery (reboot) time of 5 minutes and manual recovery time of 2 hours.

Of the 407 stratum S_3 runs, only 3 reached a down state. However, we include also the 19 inadequate runs as reaching a down state. Thus, Table 2 provides only indicative results of the unavailability ($\hat{U}$) and MTBF ($\hat{\Lambda}^{-1}$) of the MS service, and hence confidence intervals for these estimates are omitted. The results show as expected, that the two inadequate runs from stratum S_1 included with a service down time of 5 minutes, completely dominates the unavailability of the service. However, accounting for near-coincident failures may still prove important once the remaining deficiencies in the platform have be resolved. Although the results are indicative, it seems that very high availability and MTBF may be obtained for services deployed with Jgroup/ARM.

Table 2. Computed probabilities, unavailability metric and the system MTBF

	Experiment Recovery Period		Processor Recovery (5 min.)		Manual Processor Recovery (2 hrs.)	
	Processor Mean Time Between Failure (MTBF=λ^{-1}) (in days)					
	100	200	100	200	100	200
π_1	0.99999314	0.99999657	0.99975688	0.99987845	0.99412200	0.99707216
π_2	$6.855602 \cdot 10^{-6}$	$3.427801 \cdot 10^{-6}$	$2.430555 \cdot 10^{-4}$	$1.215278 \cdot 10^{-4}$	$5.833333 \cdot 10^{-3}$	$2.916667 \cdot 10^{-3}$
π_3	$4.072921 \cdot 10^{-11}$	$1.018230 \cdot 10^{-11}$	$5.595341 \cdot 10^{-8}$	$1.398835 \cdot 10^{-8}$	$4.466146 \cdot 10^{-5}$	$1.116536 \cdot 10^{-5}$
$\hat{U}$	$4.671318 \cdot 10^{-7}$	$2.335617 \cdot 10^{-7}$	$2.777102 \cdot 10^{-4}$	$1.388720 \cdot 10^{-4}$	$6.627480 \cdot 10^{-3}$	$3.323574 \cdot 10^{-3}$
$\hat{\Lambda}^{-1}$	20.3367 yrs	40.6741 yrs	-	-	-	-

6 Conclusions

This paper has presented an approach for the estimation of dependability attributes based on the combined use of fault injection and a novel post stratified sampling scheme. The approach has been used to assess and evaluate a service deployed with the Jgroup/ARM framework. The results of the experimental evaluation indicate that services deployed with Jgroup/ARM can obtain very high availability and MTBF.

Thus far, our automated fault injection tool has proved exceptionally useful in uncovering at least a dozen subtle bugs, allowing systematic stress and regression testing. In future work, we intend to improve the Jgroup/ARM framework further to reduce the number of service failures due to platform deficiencies. The approach may also be extended to provide unbiased estimators, allowing us to determine confidence intervals also for dependability attributes given enough samples visiting the down states.

References

1. Y. Amir, C. Danilov, and J. Stanton. A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, New York, June 2000.
2. J. Arlat, M. Aguera, , L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, Feb. 1990.
3. J. Arlat, M. Aguera, Y. Crouzet, J.-C. Fabre, E. Martins, and D. Powell. Experimental Evaluation of the Fault Tolerance of an Atomic Multicast System. *IEEE Transactions on Reliability*, 39(4):455–467, Oct. 1990.
4. D. Avresky, J. Arlat, J.-C. Laprie, and Y. Crouzet. Fault Injection for Formal Testing of Fault Tolerance. *IEEE Transactions on Reliability*, 45(3):443–455, Sept. 1996.

5. Ö. Babaoğlu, R. Davoli, and A. Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. *IEEE Transactions on Software Engineering*, 27(4):308–336, Apr. 2001.
6. B. Ban. JavaGroups – Group Communication Patterns in Java. Technical report, Dept. of Computer Science, Cornell University, July 1998.
7. K. Birman. The Process Group Approach to Reliable Distributed Computing. *Commun. ACM*, 36(12):36–53, Dec. 1993.
8. R. Chandra, R. M. Lefever, K. R. Joshi, M. Cukier, and W. H. Sanders. A Global-State-Triggered Fault Injector for Distributed System Evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):593–605, July 2004.
9. M. Cukier, D. Powell, and J. Arlat. Coverage Estimation Methods for Stratified Fault-Injection. *IEEE Transactions on Computers*, 48(7):707–723, July 1999.
10. P. Felber. *The CORBA Object Group Service: a Service Approach to Object Groups in CORBA*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, Jan. 1998.
11. U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proc. of the 19th Int. Symp. on Fault-Tolerant Computing*, pages 340–347, Chicago, IL, USA, June 1989.
12. K. R. Joshi, M. Cukier, and W. H. Sanders. Experimental Evaluation of the Unavailability Induced by a Group Membership Protocol. In *Proc. of the 4th European Dependable Computing Conference*, pages 140–158, Toulouse, France, Oct. 2002.
13. C. Karamanolis and J. Magee. Client-Access Protocols for Replicated Services. *IEEE Transactions on Software Engineering*, 25(1), Jan. 1999.
14. P. A. W. Lewis and E. J. Orav. *Simulation Methodology for Statisticians, Operation Analyst and Engineers*, volume 1 of *Statistics/Probability Series*. Wadsworth & Brooks/Cole, 1989.
15. H. Meling and B. E. Helvik. ARM: Autonomous Replication Management in Jgroup. In *Proc. of the 4th European Research Seminar on Advances in Distributed Systems*, Bertinoro, Italy, May 2001.
16. H. Meling and B. E. Helvik. Performance Consequences of Inconsistent Client-side Membership Information in the Open Group Model. In *Proc. of the 23rd Int. Performance, Computing, and Communications Conf.*, Phoenix, Arizona, Apr. 2004.
17. H. Meling, A. Montresor, Ö. Babaoğlu, and B. E. Helvik. Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management for Dependable Computing. Technical Report UBLCS-2002-12, Dept. of Computer Science, University of Bologna, Oct. 2002.
18. D. L. Mills. Network Time Protocol (Version 3); Specification, Implementation and Analysis, Mar. 1992. RFC 1305.
19. A. Montresor. *System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems*. PhD thesis, Dept. of Computer Science, University of Bologna, Feb. 2000.
20. P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, University of California, Santa Barbara, Dec. 1999.
21. D. Powell. Distributed Fault Tolerance: Lessons from Delta-4. *IEEE Micro*, pages 36–47, Feb. 1994.
22. Y. Ren. *AQuA: A Framework for Providing Adaptive Fault Tolerance to Distributed Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.
23. Y. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, and M. Seri. AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. *IEEE Transactions on Computers*, 52(1):31–50, Jan. 2003.

The Effectiveness of Choice of Programming Language as a Diversity Seeking Decision

Meine J.P. van der Meulen[1] and Miguel Revilla[2]

[1] Centre for Software Reliability, City University, London
`http://www.csr.city.ac.uk`
[2] Department of Applied Mathematics,
University of Valladolid, Spain
`http://www.mac.cie.uva.es/~revilla/`

Abstract. Software reliability can be increased by using a diverse pair of programs (1-out-of-2 system), both written to the same specification. The improvement of the reliability of the pair versus the reliability of a single version depends on the degree of diversity of the programs. The choice of programming language has been suggested as an example of a diversity seeking decision. However, little is known about the validity of this recommendation. This paper assesses the effect of language on program diversity.

We compare the effects of the choice of programming language as a diversity seeking decision by using programs written to three different specifications in the "UVa Online Judge". Thousands of programs have been written to these specifications; this makes it possible to provide statistical evidence.

The experiment shows that when the average probability of failure on demand (pfd) of the programs is high, the programs fail almost independently, and the choice of programming language does not make any difference. When the average pfd of the pools gets lower, the programs start to fail dependently, and the pfd of the pairs deviates more and more from the product of the pfds of the individual programs. Also, we observe that the diverse C/Pascal or C++/Pascal pairs perform as good as or better than the other possible pairs.

1 Introduction

The use of a diverse pair of programs has often been recommended to achieve high reliability [1] [2] [3] [4] [5]. Software diversity may however not lead to a dramatically high improvement. This is caused by the fact that the behaviour of the programs cannot be assumed to be independent [3] [5] [6] [7]. Two program versions written by independent teams can still contain similar programming mistakes, thus limiting the gain in reliability of the diverse pair.

In spite of this, the case for diversity for achieving high reliability remains strong. The possible gain using diversity appears to be higher than can be achieved by trying to write a high reliability single program [6].

Several techniques have been proposed to decrease the likelihood that different programs fail dependently. These are called "Diversity Seeking Decisions" in [9]. Examples are:

– **Data diversity.** Using random perturbations of inputs; using algorithm specific re-expression of inputs.
– **Design diversity.** Separate ("independent") development; diversity in programming language; diverse requirements/specifications; different expressions of identical requirements; etc.

In this paper we will concentrate on design diversity and specifically on programming language diversity. This is a potential defence against some programming slips, and provides some, limited, cognitive diversity against mistakes in higher-level problem solving; the efficacy will however depend heavily on "how different" the programming languages are.

The "UVa Online Judge"-website (`http://acm.uva.es`) provides many programs written to many specifications, and gives us the opportunity to compare diverse pairs. In this research we use the programs written in C, C++ and Pascal, written to three different specifications. Our aim is to compare the reliability performance of diverse pairs with each other and with single programs.

2 The Experiment

2.1 The UVa Online Judge

The "UVa Online Judge"-Website is an initiative of Miguel Revilla of the University of Valladolid [10]. It contains problems to which everyone can submit solutions. The solutions are programs written in C, C++, Java or Pascal. The correctness of the programs is automatically judged by the "Online Judge". Most authors submit solutions until their solution is judged as being correct. There are many thousands of authors and together they have produced more than 3,000,000 solutions to the approximately 1500 problems on the website.

In this paper we will analyse the programs written to three different problems on the website. We will submit every program to a test set, and then compare their failure behaviour.

There are some obvious drawbacks from using this data as a source for scientific analysis. First of all, these are not "real" programs: the programs under consideration solve small, mostly mathematical, problems. We have to be careful to not overinterpret the results.

Another point of criticism might be the fact that the Online Judge does not give feedback on the demand on which the program failed. This is not necessarily a drawback. It is certainly not comparable to a development process involving a programmer and a tester, because in that case there will be feedback on the input data for which the program fails. It has however similarities with a programmer's normal development process: a programmer will in spite of the fact that there are no examples of inputs for which a program fails, assume that it is not yet correct. The programmer works until he is convinced that the program is correct, based on his own analysis and testing. From this perspective, the Online Judge only confirms the programmer's intuition that the program is not yet correct. In this experiment, we circumvent this drawback by only using first submissions.

A last possible criticism on our approach is that programmers may copy each other's results. This may be true, but it is possible to limit the consequences of this plagiarism for the analyses by assuming that authors will only copy correct results from each other. For the analyses in this paper, the consequence is that we cannot trust absolute results, and we will limit ourselves to observing trends in relative performance.

2.2 Problems Selected

We selected problems from conforming to the following criteria:

- The problem does not have history, i.e. subsequent inputs should not influence each other. Of course, some programmers may implement the problem in such a way that it has history. Given our test approach, see below, we will not detect these kinds of faults.
- The problem has a limited demand space: two integer inputs.

Both restrictions lead to a reduction of the size of the demand space and this keeps the computing time within reasonable bounds (the necessary preparatory calculations for the analysis of these problems take between a day and two weeks to complete).

Below, we provide a short description of the problems, although this information is not necessary for reading this paper: we will not go into detail with respect to functionality. It gives some idea of the nature and difficulty of the problems, which is useful for interpreting our results. See the website `http://acm.uva.es` for more detailed descriptions of the problems.

The "3n+1"-Problem. A number sequence is built as follows: start with a given number; if it is odd, multiply by 3 and add 1; if it is even, divide by 2. The sequence length is the number of these steps to arrive at a result of 1. Determine the maximum sequence length for the numbers between two given integers $0 < i, j \leq 100,000$.

The "Factovisors"-Problem. For two given integers $0 \leq i, j \leq 2^{31}$, determine whether j divides factorial i.

The "Prime Time"-Problem. Euler discovered that the formula $n^2 + n + 41$ produces a prime for $0 \leq n \leq 40$; it does however not always produce a prime. Write a program that calculates the percentage of primes the formula generates for n between two integers i and j with $0 \leq i \leq j \leq 10,000$.

2.3 Running the Programs

For all problems chosen, a "demand" is a set of two integer inputs. Every program is restarted for every demand; this is to ensure the experiment is not influenced by history, e.g. when a program crashes for certain demands. We set a time limit on each demand of 200 ms. This time limit is chosen to terminate programs that are very slow, stall, or have other problems.

Table 1. Some statistics on the three problems

	3n+1			Factovisors			Prime Time		
	C	C++	Pas	C	C++	Pas	C	C++	Pas
Number of authors	5897	6097	1581	212	582	71	467	884	183
First attempt correct	2483	2442	593	113	308	42	356	653	127
First version completely incorrect	723	761	326	27	97	9	93	194	49

Every program is submitted to a series of demands. The outputs generated by the programs are compared to each other. Programs that produce exactly the same outputs form an "equivalence class". These equivalence classes are then converted into score functions. A score function indicates which demands will result in failure. The difference between an equivalence class and its score function is that programs that fail in different ways (i.e. different, incorrect outputs for the same demands) are part of different equivalence classes; their score functions may however be the same. The score functions are used for the calculations below.

For all three problems, we chose the equivalence class with the highest frequency of occurrence as the oracle, i.e. the version giving all the correct answers.

"3n+1" and "Factovisors" were run using the same set of demands: two numbers between 1 and 50, i.e. a total of 2500 demands. In both cases the outputs of the programs were deemed correct if they exactly match those of the oracle.

"Prime Time" was run using a first number between 0 and 79, and a second number between the first and 79, i.e. a total of 3240 demands. The outputs of the programs were deemed correct if they were within 0.01 of the output of the oracle, thus allowing for errors in round off (the answer is to be given in two decimal places).

Table 1 gives some statistics on the problems.

3 Effectiveness of Diversity

Two of the most well known probability models in this domain are the Eckhardt and Lee model [3] and the Littlewood and Miller extended model [7]. Both models assume that:

1. Failures of an individual program are deterministic and a program version either fails or succeeds for each input value x. The failure set of a program π can be represented by a "score function" $\omega(\pi, x)$ which produces a zero if the program succeeds for a given x or a one if it fails.
2. There is randomness due to the development process. This is represented as the random selection of a program, Π, from the set of all possible program versions that can feasibly be developed and/or envisaged. The probability that a particular version π will be produced is $P(\Pi = \pi)$. This can be related to the relative numbers of programs in the pools.
3. There is randomness due to the demands in operation. This is represented by the random occcurrence of a demand, X, from the set of all possible

demands. The probability that a particular demand will occur is $P(X = x)$, the demand profile. In this experiment we assume a contiguous demand space in which every demand has the same probability of occurring.

Using these model assumptions, the average probability of a program version failing on a given demand x is given by the difficulty function, $\theta(x)$, where:

$$\theta(x) = \sum_{\pi} \omega(\pi, x) P(\Pi = \pi) \tag{1}$$

The average probability of failure on demand (pfd) of a randomly chosen single program version, can be computed using the difficulty function and the demand profile, in our case for two program versions, Π_A and Π_B:

$$\text{pfd}_A := \sum_x \theta_A(x) P(X = x), \quad \text{pfd}_B := \sum_x \theta_B(x) P(X = x) \tag{2}$$

The Eckhardt and Lee model assumes similar development processes for the two programs A and B and hence identical difficulty functions: $\theta_A(x) = \theta_B(x)$. So the average pfd for a pair of diverse programs (assuming the system only fails when both versions fail, i.e. a 1-out-of-2 system) would be:

$$\text{pfd}_{AB} := \sum_x \theta_A(x)\theta_B(x) P(X = x) = \sum_x \theta_A(x)^2 P(X = x) \tag{3}$$

And:

$$\text{pfd}_{AB} = \text{pfd}_A^2 + var_X(\theta_A(X)) \tag{4}$$

The Littlewood and Miller model does not assume that the development processes are similar, and thus allows the difficulty functions for the two versions, $\theta_A(x)$ and $\theta_B(x)$, to be different. Therefore, the probability of failure of a pair remains:

$$\text{pfd}_{AB} := \sum_x \theta_A(x)\theta_B(x) P(X = x) \tag{5}$$

And:

$$\text{pfd}_{AB} = \text{pfd}_A.\text{pfd}_B + cov_X(\theta_A(X),\ \theta_B(X)) \tag{6}$$

In this experiment, we wish to investigate the reliability improvement gained by choosing different programming languages for the programs in the pair, and we therefore need to use the Littlewood and Miller model.

First, we establish pools of programs, each pool containing programs in C, C++ or Pascal. For comparison of the individual programs and pairs, we need pools with the same pfd. To manipulate the pfd of the pools, we remove programs from them, starting with those with the highest pfd, until the average pfd of the pool has the desired value. This is a possible way of simulating testing of the programs; the tests remove the programs with the highest pfd first. Pools with the same pfd could then be assumed to have undergone the same scrutiny of testing.

We select a first program from one of the pools. Then we select a second program from a pool, and calculate the ratio of the pfd of the first program and the pfd of the pair:

$$R = \frac{\text{pfd}_A}{\text{pfd}_{AB}} = \frac{\sum_x \theta_A(x)P(X=x)}{\sum_x \theta_A(x)\theta_B(x)P(X=x)} \tag{7}$$

We do so for varying values of the pfd of the pools. The varying pfd is shown on the horizontal axis in the graphs.

Figures 1, 2 and 3 show these ratios for the three problems for different choices of the programming language of the first program. The graphs show R on the vertical axes on a logarithmic scale, because we are interested in the reliability improvement; with a logarithmic scale, equal improvements have equal vertical distance.

4 Analysis and Discussion

All graphs clearly show that for pairs of programs with higher pfds (pfd $> 10^{-1.5}$) the choice of programming language does not make any difference. The pfd of the pair is fairly close to the product of the pfds of the individual programs. This indicates that the programs fail almost independently.

For the lower pfds (pfd $< 10^{-2}$) the pfd of the pair deviates more and more from the product of the pfds of the individual programs, the programs fail dependently. The reliability improvement reaches a "plateau" at between one to two orders of magnitude better than a single version. This is in accordance with the generally accepted assumption that the gain from redundancy is limited, and is certainly not simply the product of the pfds of the individual programs (e.g. in IEC61508 [13] the reliability improvement one can claim for applying redundancy is one SIL, i.e. a factor of 10; Eckhardt and Lee reach a comparable conclusion in [4]).

If we now compare the effect of choice of programming language in the pairs, we can observe that the C/Pascal and the C++/Pascal pairs almost always outperform the other pairs, most notably also the C/C++ pairs. The effect is most clearly visible in the "Factovisors"-problem, and in the middle region ($10^{-3} <$ pdf $< 10^{-2}$) of the "3n+1"-problem. It is also visible in the "Prime Time"-problem, but in this case it is only visible in a small, unreliable, region of the graph. This graph is unreliable for pfd $< 10^{-2.5}$, because the pool of Pascal programs is almost empty.

The Littlewood and Miller model provides a description of diversity, in which the reliability change compared to the independence assumption is given by a covariance term, see Equation 6. Since the model cannot predict the shapes of the difficulty functions in C, C++ or Pascal, the model can not predict how large this change will be. The model however provides an intuition that when programmers make different faults in different programming languages their respective difficulty functions will be different. These differences could then lead to a better performance of diverse language pairs.

Analysis of the differences in programming faults of the “3n+1”-problem shows that faults in programming “for”-loops in Pascal are rare compared to C and C++. This accounts for the reliability improvement of Pascal/C and Pascal/C++ pairs in the middle pfd-region.

In the low-pfd region, $\text{pfd} < 10^{-4}$, we observe for “3n+1” that the reliability improvements of the pairs become approximately the same, whereas for “Factovisors” the diverse language pairs (C/Pascal and C++/Pascal) show an enormous improvement, even approaching infinity for the lowest pfd-values. This observation gives rise to some thoughts[1]

First, these observations in the low-pfd region confirm the theoretical result of the Littlewood and Miller model that the reliability of a diverse pair can be better than under the independence assumption.

Second, why do we observe this effect? As explained above, the pfd of a pool is established by subsequently removing the most unreliable programs. For low pfd-values this approach leads to a monoculture, and in the end only few different program behaviours will be present in the pool. In the case of “3n+1” these program behaviours happen to be roughly the same. In the case of “Factovisors” however, the last programs in the Pascal pool fail on different demands than those left in the C and C++ pools, thus leading to an enormous improvement in the reliability of the pair.

This result should be considered with care, because it could be an artefact of our experiment, caused by the way in which we establish a given pfd for a pool. On the other hand, a normal debugging process will have a similar effect: eliminating some behaviours, starting with the most unreliable ones, thus eventually also leading to a monoculture of behaviours.

5 Conclusion

We analysed the effect of the choice of programming language in diverse pairs using three different problems from the “UVa Online Judge”. The results seem to indicate that diverse language pairs outperform other pairs, but the evidence is certainly not strong enough for a definite conclusion. Analysis of more problems could help to strengthen the evidence and also to identify the factors that influence the gain possibly achieved by diversity of programming language.

Acknowledgement

This work is supported by EPSRC/UK in the framework of the *Interdisciplinary Research Collaboration in Dependability*-Project (DIRC).

[1] It has to be noted here that the amount of Pascal programs in this pfd-region for “Factovisors” is rather low, and the graph has to taken cum grano salis.

References

1. Brilliant, S.S., J.C. Knight, N.G. Leveson, *Analysis of Faults in an N-Version Software Experiment*, IEEE Transactions on Software Engineering, SE-16(2), pp. 238-47, February 1990.
2. Voges, U., *Software diversity*, Reliability Engineering and System Safety, Vol. 43(2), pp. 103-10, 1994.
3. Eckhardt, D.E., L.D. Lee, *A Theoretical Basis for the Analysis of Multi-Version Software Subject to Coincident Errors*, IEEE Transactions on Software Engineering, Vol. SE-11(12), pp. 1511-1517, December 1985.
4. Eckhardt, D.E., A.K. Caglayan, J.C. Knight, L.D. Lee, D.F. McAllister, M.A. Vouk, J.P.J. Kelly, *An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability*, IEEE Transaction on Software Engineering, Vol. 17, No. 7, July 1991.
5. Knight, J.C., N.G. Leveson, *An Experimental Evaluation of the Assumption of Independence in Multiversion Programming*, IEEE Transaction on Software Engineering, Vol. SE-12(1), pp. 96-109, 1986.
6. Hatton, L., *N-Version Design Versus One Good Version*, IEEE Software, 14, pp. 71-6, 1997.
7. Littlewood, B., D.R. Miller, *Conceptual Modelling of Coincident Failures in Multiversion Software* IEEE Transactions on Software Engineering, Vol. 15, No. 2, pp. 1596-1614, December 1989.
8. Lyu, M.R., *Software Reliability Eningeering*, McGraw Hill, 1995.
9. Popov, P., L. Strigini, A. Romanovsky, *Choosing Effective Methods for Design Diversity - How to Progress from Intuition to Science*, In: Proceedings of the 18th International Conference, SAFECOMP '99, Lecture Notes in Computer Science 1698, Toulouse, 1999.
10. Skiena, S., M. Revilla, *Programming Challenges*, Springer Verlag, March 2003.
11. Lee, P.A., T. Anderson, *Fault Tolerance; Principles and Practice*, Dependable Computing and Fault-Tolerant Systems, Vol. 3, Second, Revised Edition, 1981.
12. Chen, L., A. Avizienis, *N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation*, Digest of 8th Annual International Symposium on Fault Tolerant Computing, Toulouse, France, pp. 3-9, June 1978.
13. IEC, IEC61508, *Functional Safety of E/E/PE safety-related systems*, Geneva, 2001-2.

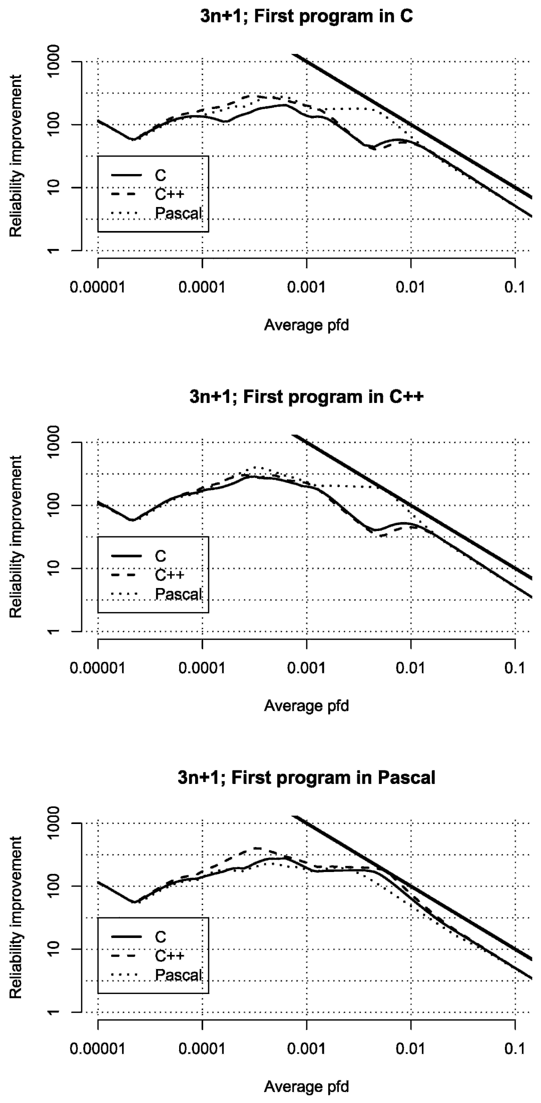


Fig. 1. These graphs show the reliability improvement of a pair of programs over a single version for the "3n+1"-problem. The horizontal axis gives the average pfd of the pools of programs involved in the calculation. In every graph, the programming language of the first program is given. The curves show the reliability improvement for the different possible choices of the programming language for the second program as function of the average pfd of the pools. The diagonal in the graphs shows the reliability achievement if the programs' behaviours were independent

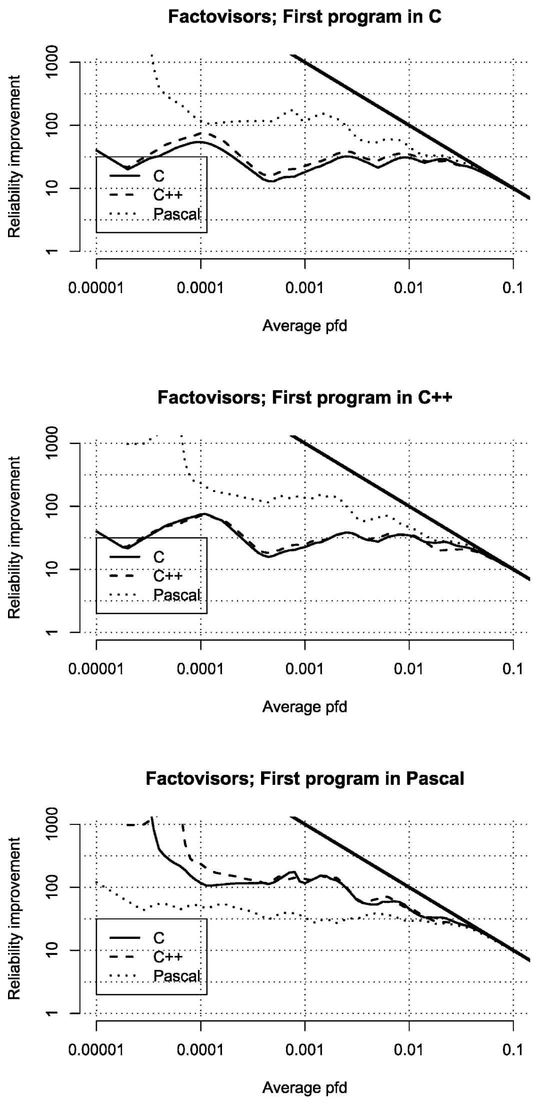


Fig. 2. The same graphs, for the “Factovisors”-problem

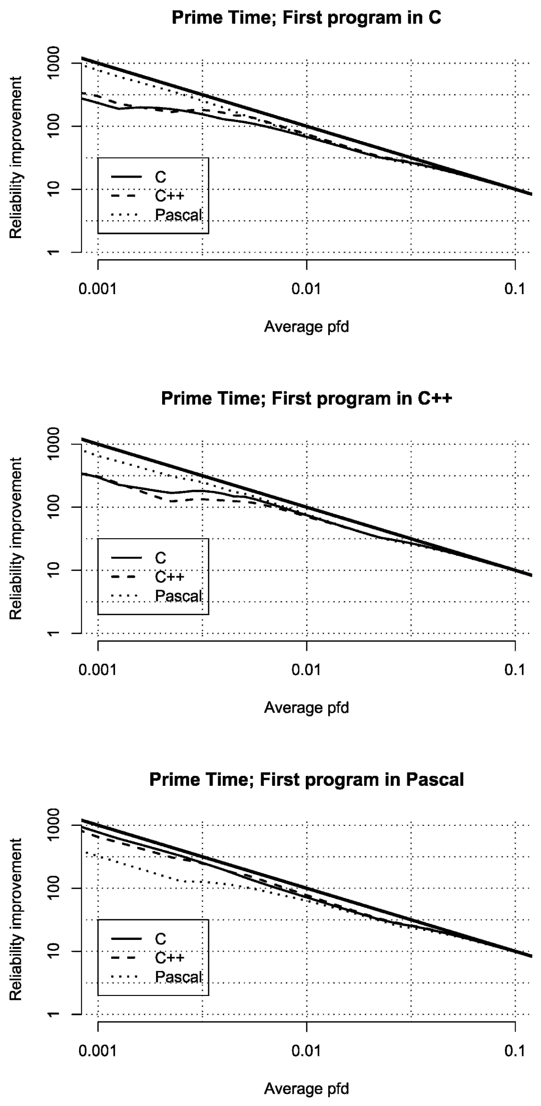


Fig. 3. The same graphs, for the "Prime Time"-problem. These graphs do not show the results for pfds below $10^{-2.5} = 3.2 \times 10^{-3}$, because the pool of Pascal programs is almost empty

Formal Safety Analysis of a Radio-Based Railroad Crossing Using Deductive Cause-Consequence Analysis (DCCA)

Frank Ortmeier, Wolfgang Reif, and Gerhard Schellhorn

Lehrstuhl für Softwaretechnik und Programmiersprachen,
Universität Augsburg, D-86135 Augsburg
{ortmeier, reif, schellhorn}@informatik.uni-augsburg.de

Abstract. In this paper we present the formal safety analysis of a radio-based railroad crossing. We use deductive cause-consequence analysis (DCCA) as analysis method. DCCA is a novel technique to analyze safety of embedded systems with formal methods. It substitutes error-prone informal reasoning by mathematical proofs. DCCA allows to rigorously prove whether a failure on component level is the cause for system failure or not. DCCA generalizes the two most common safety analysis techniques: failure modes and effects analysis (FMEA) and fault tree analysis (FTA).

We apply the method to a real world case study: a radio-based railroad crossing. We illustrate the results of DCCA for this example and compare them to results of other formal safety analysis methods like formal FTA.

Keywords: Formal methods, safety critical systems, safety analysis, failure modes and effects analysis, fault tree analysis, dependability.

1 Introduction

The central question of safety analysis is to determine what components of a safety-critical system must fail to allow the system to cause damage. Most safety analysis techniques rely only on informal reasoning which depends heavily on skill and knowledge of the safety engineer. Some of these techniques have been formalized.

In this paper we present a new safety analysis technique: Deductive Cause-Consequence Analysis (DCCA). This technique is a formal generalization of well-known safety analysis methods like FMEA [10], FMECA [4] and FTA [3]. The logical framework of DCCA may be used to rigorously verify the results of these informal safety analysis techniques. It is also strictly more expressive (in terms of what can by analyzed) than traditional FMEA. We show, that the results of DCCA have the same semantics as those of formal FTA [11]. Because of this DCCA may be used to verify fault trees without formalizing inner nodes of the tree.

In Sect. 2 the semantics of DCCA is presented. A comparison to FMEA and FTA is done in Sect. 3. We illustrate the application of DCCA and report on practical experiences in Sect. 4. In Sect. 5 some related approaches are discussed and Sect. 6 summarizes the results and concludes the paper.

2 DCCA

In this section we describe the formal semantics of DCCA. The formalization is done with Computational Tree Logic(CTL) [5]. We use finite automata as system models. The use of CTL and finite automata allows to use powerful model checkers like SMV [8] to verify the proof obligations.

In the following we assume that a list of hazards on system level and a list of possible basic component failures modes is given. Both data may be collected by other safety analysis techniques like failure-sensitive specification [9] or HazOp [6]. We assume that system hazards H and primary failures δ are described by predicate logic formula. This is true for most practical problems. We call the set of all failure predicates Δ.

2.1 Failure/Hazard Automata

For formal safety analysis failure modes must be explicitly modeled. We divide the modeling into two steps. First we model the occurrence pattern of the failure mode and second we model the failure mode itself. By "occurrence pattern" we understand how and when the failure mode occurs. For example does the failure mode occur indeterministically (like packet loss in IP traffic) or does it occur once and forever (like a broken switch) or does it occur only during certain time intervals (like until the next maintenance). To model this we use failure automata. Figure 1 shows two such failure automata.

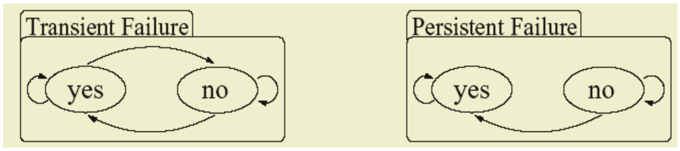


Fig. 1. Failure automata for transient and persistent failures

The left automaton models a transient failure which can indeterministically occur and disappear. The right one models a persistent failure, which happens once and stays forever (e.g. a broken relay). Maintenance etc. may be modeled analogously. Failure predicates δ are then defined as "failure automaton for failure mode $\tilde{\delta}$ in state *yes*". For readability the symbol δ is used for both the predicate and the automaton describing the occurrence pattern.

The second step is to model the direct effects of failure modes. This is usually done by adding transitions to the model with conditions of the form $\varphi \wedge \delta$. This mean these additional transitions – which reflect erroneous behavior – may only be taken, when a failure automaton is in state *yes* i.e. when a failure occurs.

A similar approach may be used to define predicates for system hazards. If the system hazard can not be describe by a predicate logic formula directly, then often an observer automaton may be implemented such that whenever the automaton is in an accepting state, the hazard has occurred before [12]. This allows to describe the hazard as predicate logic formula on the states of the observer automaton. However in practical applications hazards may usually be described by predicate logic formulas.

2.2 Critical Sets

The next step is to define a temporal logic property which says, whether a certain combination of failure modes may lead to the hazard or not. This property is called *criticality* of a set of failure modes.

Definition 1. *critical set / minimal critical set*
For a system SYS and a set of failure modes Δ a subset of component failures $\Gamma \subseteq \Delta$ is called critical for a system hazard, which is described by a predicate logic formula H if

$$SYS \models \mathbf{E}(\overline{\lambda}\ \mathbf{until}\ H) \ \ \textit{where}\ \overline{\lambda} := \bigwedge_{\delta \in (\Delta \setminus \Gamma)} \neg\, \delta$$

We call Γ a minimal critical set if Γ is critical and no proper subset of Γ is critical.

Here, $\mathbf{E}(\varphi\ \mathbf{until}\ \psi)$ denotes the existential CTL-UNTIL-operator. It means there exists a path in the model, such that φ holds until the property ψ holds. The property *critical set* translates into natural language as follows: there exists a path such that the system hazard occurs without the previous occurrence of any failures except those which are in the critical set. In other words this means, it is possible that the systems fails, if only the component failures in the critical set occur. Intuitively, criticality is not sufficient to define a cause-consequence relationship. It is possible that a critical set includes failure modes, which have nothing to do with the hazard.

Therefore, the notion *minimal critical set* also requires that no proper subset of is critical. Minimal critical sets really describe what one would expect for a cause-consequence relationship in safety analysis to hold: the causes may - but not necessarily - lead to the consequence and second all causes are necessary to allow the consequence to happen. So the goal of DCCA is to find minimal critical sets of failure modes. Testing all sets by brute force would require an effort exponential in the number of failure modes. However, DCCA may be used to formally verify the results of informal safety analysis techniques. This reduces the effort of DCCA a lot, because the informal techniques often yield good "initial guesses" for solutions. Note that the property critical is monotone with respect to set inclusion i.e. $\forall \Gamma_1, \Gamma_2 \subseteq \Delta : \Gamma_1 \subseteq \Gamma_2 \Rightarrow$ (Γ_1 is critical set $\Rightarrow \Gamma_2$ is critical set). This helps to reduce proof efforts a lot.

3 Comparison to Other Safety Analysis Methods

DCCA formalizes different methods of formal safety analysis in a generic way. We can identify different cases according to the number of elements in the set of failure modes being analyzed and relate them to other existing safety analysis techniques.

$|\Gamma| = 0$

If the empty set of failure modes is examined, then the proof obligation of minimal criticality corresponds to the verification of functional incorrectness. Minimality is of course satisfied (the empty set does not have real subsets). The property of criticality states that there "exists a path where no component fails but eventually the hazard occurs" (in CTL: $EF\ H$). This is the negation of the standard property of functional correctness "on all paths where no component fails, the hazard will globally not occur" (in CTL: $AG\ \neg H$). In other words, if the empty set can be proven to be a critical set, then the system has design errors and is functionally incorrect.

$|\Gamma| = 1$

The analysis of single failure modes corresponds to traditional FMEA. Traditional FMEA analyzes the effects of a component failure mode on the total system in an informal manner. If the failure modes appears to be safety critical than this cause-consequence relationship is noted as one row of a (FMEA) spreadsheet. If a singleton set is minimal critical for a hazard H, then a correct FMEA must list the hazard H as effect of the analyzed failure mode. Note that functional correctness is a pre-condition for formal FMEA. If the system is not functionally correct, then there will be no singleton sets of failure modes which are minimal critical.

$|\Gamma| > 1$

This is a true improvement to FMEA. Combinations of component failure modes are traditionally only examined by FTA. FTA analyzes top-down the reasons of system failure. Cause and consequence are linked by certain *gates*. The *gates* of a fault tree state if all causes (AND-gate) or any of the causes (OR-gate) are necessary to allow the consequence to occur. Iteration builds a tree like structure where the root is the system hazard and the leaves are component failures.

The result of FTA is a set of so called *minimal cut sets*. These sets may be generated automatically from the structure of the tree. Each *minimal cut set* describes a set of failure modes, which together may make the hazard happen. This corresponds to the definition of *minimal critical sets* obtained by DCCA. So FTA may be seen as a special case of DCCA. An introduction to FTA may be found in [3].

FTA has been enhanced with formal semantics. Formal FTA [12] allows to decide whether failure modes have been forgotten or not. The idea is to assign a temporal logic formula to each gate. If this formula is proven correct for the system, then the gate is *complete*. This means no causes have been forgotten. An example is given in figure 2.

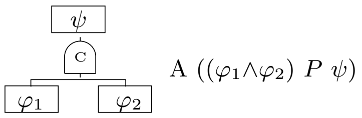


Fig. 2. Fault tree gate and formalization

The figure shows a synchronous cause-consequence AND-gate. The semantics is that both reasons φ_1 and φ_2 must occur simultaneously, before the consequence ψ may occur. Here, $A(\varphi\ P\ \psi)$ denotes the derived CTL-operator *PRECEDES*, which is defined as $\neg E(\neg\varphi\ \mathit{until}\ (\psi \wedge \neg\varphi))$. Informally *PRECEDES* means that whenever ψ holds, φ must have happened before.

Altogether formal FTA distinguishes 7 different types of gates, which reflect temporal ordering, environment constraints and synchronous vs. asynchronous dependencies between cause and consequence. A detailed description of formal FTA may be found in [13].

One of the main results of FTA is the minimal cut set theorem. This theorem states that for a complete fault tree the prevention of only one failure mode of every minimal cut set, assures that the system hazard will never occur. A fault tree is called complete, if all its gates have been proven to be complete.

DCCA may be used to verify the completeness of a fault tree analysis as well. To apply DCCA to FTA we must first introduce the notion of a *complete* DCCA. We call a DCCA complete if all minimal critical sets have been identified.

If a DCCA has been shown to be complete, then it is proven that the minimal critical sets of the DCCA have the same meaning as the minimal cut sets of a fault tree done with formal FTA. In particular the following theorem holds:

Theorem 1. *Minimal critical sets*
For a complete DCCA prevention of one element of every minimal critical set will prevent the hazard H from occurring.

The proof of this theorem is very easy. The statement may be directly derived from the definition of minimal critical sets and the semantics of CTL. In the following SYS denotes a CTL model, s_0 is the initial state of the model, minimal critical sets are called Γ, the set of all minimal critical sets is Ω and individual failure modes are labeled with δ. We define (CTL*-) formulas $[\Gamma]$ for "there exist a path in the system, such that eventually a critical set of failures Γ occurs" and $[[\Omega]]$ for "there does not exist a path, such that any of the minimal critical sets $\Gamma \in \Omega$ occurs".

$$[\Gamma] := E \bigwedge_{\delta \in \Gamma} F\delta$$

$$[[\Omega]] := \bigwedge_{\Gamma \in \Omega} \neg[\Gamma] = \bigwedge_{\Gamma \in \Omega} \neg E \bigwedge_{\delta \in \Gamma} F\delta$$

With this abbreviations theorem 1 rewrites to $SYS \models [[\Omega]] \Rightarrow SYS \models AG\neg H$. In the following traces of system SYS are called σ and states s_i. The proof for theorem 1 is then as follows:

Proof.

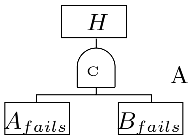
$$\begin{aligned}
Assume: \quad & SYS \not\models AG\neg H \\
& \Leftrightarrow SYS, s_0 \not\models AG\neg H \\
& \Leftrightarrow SYS, s_0 \not\models \neg EF\neg\neg H \\
& \Leftrightarrow SYS, s_0 \models EFH \\
& \Leftrightarrow SYS, s_0 \models E(trueUH) \\
& \Leftrightarrow \exists \sigma = (s_0, s_1, ...) \in SYS : SYS, \sigma \models (trueUH) \\
& \Leftrightarrow \exists i: \; SYS, s_i \models H \wedge \forall j < i: \; SYS, s_j \models true \\
Let \quad & \Gamma_\dagger := \{\delta \in \Omega \mid \exists\, j < i \; : SYS, s_j \models \delta\} \\
& \Rightarrow SYS, \sigma \models (\overline{\Gamma_\dagger} UH) \\
& \Leftrightarrow \Gamma_\dagger \text{ is critical set} \\
& \Rightarrow \exists \tilde{\Gamma}_\dagger \subseteq \Gamma_\dagger : \; \tilde{\Gamma}_\dagger \text{ is minimal critical set} \\
Furthermore \quad & SYS, \sigma \models \bigwedge_{\delta_j \in \Gamma_\dagger} F\delta_j \\
& \Rightarrow SYS, \sigma \models \bigwedge_{\delta_j \in \tilde{\Gamma}_\dagger} F\delta_j \\
& \Rightarrow SYS, s0 \models E \bigwedge_{\delta_j \in \tilde{\Gamma}_\dagger} F\delta_j \\
& \Leftrightarrow SYS \models [\tilde{\Gamma}_\dagger] \\
& \Rightarrow SYS \not\models [[\Omega]], \; as \; \tilde{\Gamma}_\dagger \in \Omega, \text{ because DCCA is complete} \\
& \Rightarrow \lightning \quad \square
\end{aligned}$$

For a complete DCCA even the following, stronger result holds:

$$SYS, s_0 \models A((\bigwedge_{\Gamma_i \in \Omega} \neg \bigwedge_{\delta_j \in \Gamma_i} F\delta_j) \rightarrow G\neg H)$$

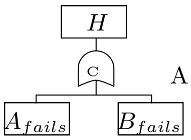
This is the same property that holds for formal fault tree analysis with the semantics of [12]. However there is a difference as DCCA is more precise. Formal FTA may yield weaker cut sets than DCCA. For example, assume a systems SYS has two redundant units A and B. The hazard H may only occur if both units fail. So the system has only one minimal critical set of failures: "*A fails*

AND B fails". As an intuitively consequence the fault tree in figure 3 is correct and the fault tree in figure 4 is incorrect.



A $((A_{fails} \wedge B_{fails})$ P H)

Fig. 3. Correct fault tree



A $((A_{fails} \vee B_{fails})$ P H)

Fig. 4. Incorrect fault tree

With formal FTA both fault trees may be proven to be complete, as both formulas may be proven correct for this system. But the fault tree of figure 3 will yield only one minimal cut set $\Gamma = \{A_{fails}, B_{fails}\}$ while the fault tree of figure 4 will yield two singleton minimal cut sets $\Gamma_1 = \{A_{fails}\}$ and $\Gamma_2 = \{B_{fails}\}$. It is not possible to distinguish the two fault trees with formal FTA. On the other hand DCCA will discover that $\{A_{fails}\}$ resp. $\{B_{fails}\}$ is not critical, because the formula correspondind DCCA formula evaluates to false and thus the sets are not critical. The set $\Gamma = \{A_{fails}, B_{fails}\}$ can be proven to be minimal critical.

The reason for this difference is that formal fault tree semantics does NOT require that all causes must occur before the consequence, but only that prevention of causes prevents the consequence. Here, DCCA yields more precise results than formal FTA. A second advantage is that DCCA does not require inner nodes to be formalized. This is a big advantage in practical applications. Inner nodes are often very hard to formalize. For example an inner node of the fault tree of the example of Sect. 4 is "Release sent and barriers opening". Since "Release *sent*" refers to the past, this is not directly expressible in CTL. This problem was discovered in many case studies and was one of the motivating factors that led to the development of DCCA.

A problem of showing completeness of DCCA is of course the exponential growth of the number of proof obligations. However, only big minimal critical sets will result in a lot of proof effort. In many real applications minimal critical sets are rather small. In addition, informal safety analysis helps to find candidates for minimal cut sets in advance. FTA is one possibility, FMEA is another. This reduces the combinatorial effort of checking all possible sets of failure modes a lot. Finally, monotony of the property *critical* may be exploited; if e.g. a singleton set is minimal critical, then other minimal critical sets must not contain this element.

4 Application

As an example for the application of DCCA we present an analysis of a radio-based railroad crossing. This case study is the reference case study of the german research councils (DFG) priority program 1064. This programs aims at bringing together field-tested engineering techniques with modern methods of the domain of software engineering.

The German railway organization, Deutsche Bahn, prepares a novel technique to control railroad crossings: the decentralized, radio-based railroad crossing control. This technique aims at medium speed routes, i.e. routes with maximum speed of 160 km/h. An overview is given in [7].

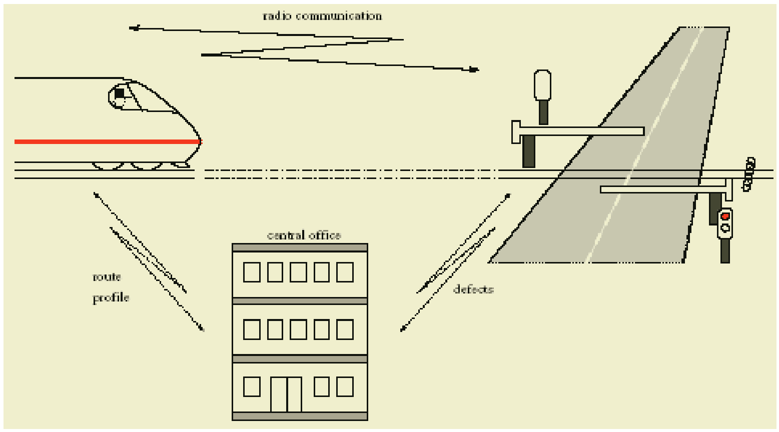


Fig. 5. Radio-based railroad crossing

The main difference between this technology and the traditional control of railroad crossings is that signals and sensors on the route are replaced by radio communication and software computations in the train and railroad crossing. This offers cheaper and more flexible solutions, but also shifts safety critical functionality from hardware to software.

Instead of detecting an approaching train by a sensor, the train computes the position where it has to send a signal to secure the level crossing. To calculate the activation point the train uses data about its position, maximum deceleration and the position of the crossing. Therefore the train has to know the position of the railroad crossing, the time needed to secure the railroad crossing, and its current speed and position. The first two items are memorized in a data store and the last two items are measured by an odometer. For safety reasons a safety margin is added to the activation distance. This allows compensating some deviations in the odometer. The system works as follows:

The train continuously computes its position. When it approaches a crossing, it broadcasts a 'secure'-request to the crossing. When the railroad crossing receives the command 'secure', it switches on the traffic lights, first the 'yellow' light, then the 'red' light, and finally closes the barriers. When they are closed, the railroad crossing is 'secured' for a certain period of time. The 'stop' signal on the train route, indicating an insecure crossing, is also substituted by computation and communication. Shortly before the train reaches the 'latest braking point' (latest point, where it is possible for the train to stop in front of the crossing), it requests the status of the railroad crossing. When the crossing is secured, it responds with a 'release' signal which indicates, that the train may pass the crossing. Otherwise the train has to brake and stop before the crossing. The railroad crossing periodically performs self-diagnosis and automatically informs the central office about defects and problems. The central office is responsible for repair and provides route descriptions for trains. These descriptions indicate the positions of railroad crossings and maximum speed on the route. The safety goal of the system is clear: it must never happen, that the train is on the crossing and a car is passing the crossing at the same time. A well designed control system must assure this property at least as long as no component failures occur. The corresponding hazard H is "a train passes the crossing and the crossing is not secured". This is the only hazard which we will consider in this case study

4.1 Formal Model

We now give a brief description of the formal system model. We used SMV [8] as model checker. Altogether the system consists of 16 automata: two automata modeling the control of the crossing and the train, five timer automata, six failure automata and three automata modeling the physics of the train. Altogether the model has roughly 1100 states. In the following we give brief descriptions of the most interesting automata. For better readability - we use a graphical notation instead of SMV input language.

Primary Failure and Hazards. We will now briefly explain the analyzed failure modes and hazards and how they are modeled. The modeling of failure modes generally splits into two different tasks: the modeling of the occurrence pattern and the direct effect of the failure mode. The occurrence pattern describes, when and how the failure occurs resp. when it does not occur. We model occurrence patterns with failure automata (see Sect. 2.1).

In the following we give a summary of the failure modes, which we analyzed. In this example only one hazard is interesting i.e. the train passes an insecure crossing. We call this hazard collision H_{Col}. This is modeled by the following formula:

$$H_{Col} := Pos \leq Pos_{ds} \wedge Pos + Speed > Pos_{ds} \wedge \neg Crossing = closed$$

In this formula Pos_{ds} is an abbreviation for the position of the crossing (ds = danger spot). It describes the location of the crossing. H_{Col} evaluates to true, iff the train passes the crossing and the barriers are not closed. We investigated the following six different types of component failures:

- **Failure of the brakes:** *error_brake* -This error describes the failure of the brakes. It has direct effects on automaton *Dec.*
- **Failure of the communication:** *error_comm* - This error describes the failure of the radio communication. It has direct effects on automata $timer_{close_{rcv}}$, $timer_{status_{rcv}}$, $timer_{ack_{rcv}}$.

- **Failure of the barriers closed sensor:** *error_closed* - This error describes that the crossing signals *closed*, although it is not closed. It has direct effects on automaton *crossing.*
- **Failure of the barriers' actuator:** *error_actuator* - This error describes that the actuator of the crossing fails. It has direct effects on automaton *crossing.*
- **Failure of the train passed sensor:** *error_passed* - This error describes that the sensor detecting trains which passed the crossing fails. It has direct effects on automaton *crossing.*
- **Deviation in the odometer:** *error_odo* - This error describes that the odometer does not give 100% precise data. It has direct effects on automaton *train_control.*

The occurrence of each of these failure modes is modeled by a failure automaton. All failure modes - except *error_actuator* - are assumed to be transient. As abbreviation we write $error\_brake$ for $error\_brake = yes$.

Model of the Crossing. The automaton in figure 6 shows the model of the crossing. Initially the barriers are *opened.* When the crossing receives a close request from an arriving train - i.e. condition *comm_close_rcv* becomes true, the barriers start *closing.* This process takes some time. This is modeled by timer automaton *timer_closing.* After a certain amount of time the barriers are *closed.* They will remain closed until the train has passed the crossing (detected by a sensor). The barriers reopen automatically after a defined time interval. This is a standard procedure in railroad organization, as car drivers tend to ignore closed barriers at a railroad crossing if the barriers are closed too long. So it is better to reopen the barriers, than having car drivers slowly driving around the closed barriers. The reopening is modeled using another timer automaton *timer_closed.*

A faulty signal from the sensor, which detects when the train has passed the crossing will also open the crossing. This is modeled by *error_passed = true.* The barriers may get *stuck*, if the actuator fails (*error_actuator*).

Model of the Train Control. The train control supervises the position of the train, issues closing requests to the crossing and ultimately decides, if an emergency stop is necessary or not. The train control is implemented in software on-board the train. The formal model is given in figure 7. Starting from its initial state *idle* the automaton goes into state *wfc* ('wait for close'), if the train approaches the crossing (*pos_close_reached*). Simultaneously the train sends a signal requesting to close the barriers.

Some time later the train sends a status request message to the crossing and waits for an answer (state *wfs* - 'wait for status answer'). If the positive answer reaches the train in time, then the control allows passing the crossing and enters state *finish.* If no acknowledge is received, the control issues an emergency stop (state *brake*). In this case the crossing must be secured manually, before the train may pass the crossing.

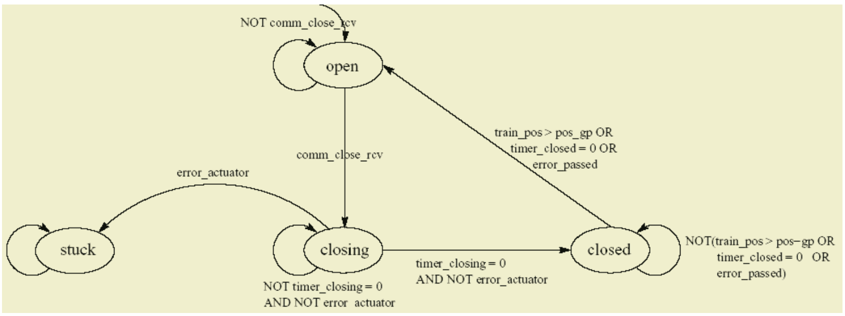


Fig. 6. Model of the crossing

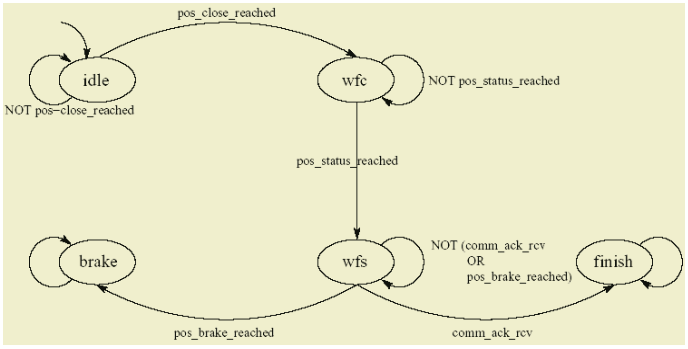


Fig. 7. Model of the train control

The three important predicates *pos_close_reached, pos_status_reached* and *pos_brake_reached* are computed from position and speed data. Possible errors result by deviation of the speed sensor of the train (called odometer). This means the control might calculate braking distances etc. wrongly.

Model of the Communication. It is part of the case study to take communication delay explicitly into account. To model this three additional timer automata ($timer_{close_{rcv}}, timer_{status_{rcv}}, timer_{ack_{rcv}}$) are built. Timer automata allow to delay certain transitions for n steps and have respectively n states. If, for example, the train sends a close signal to the crossing, timer $timer_{close_{rcv}}$ starts a countdown (in every step it makes transition from state n to state n-1). When finally $timer_{close_{rcv}}$ reaches state 0, the condition *comm_close_rcv* becomes true which means close signal is received by the crossing. Failure of communication prevents the signal *comm_close_rcv* from being received. The others communication request are modeled analogously.

Model of the Train. The physical train is modeled by three important properties: position, speed and acceleration. To improve readability, we give textual representation of these automata.

The position of the train is given as an integer value between 0 and Pos_{Max}. The automaton modeling the position of the train is defined as follows:

$$\begin{aligned} Pos &: 0..Pos_{Max} \\ Pos_{t=0} &:= 0 \\ Pos_{t=n+1} &:= \begin{cases} 0 \; or \; 1, \; if \; Pos_{t=n} = 0 \\ Pos_{t=n} + Speed_{t=n}, & if \; Pos_{t=n} + Speed_{t=n} \leq Pos_{Max} \\ Pos_{Max}, & otherwise \end{cases} \end{aligned}$$

This automaton models monotone movement of the train. State *Pos = 0* is an abstraction for “the train has not reached the crossing”. At every step in time it is possible that the train either stays absent *Pos = 0* or enters the region in front of the crossing *Pos = 1*. Between 1 and Pos_{max} the train moves according to its speed. When the train reaches the upper bound of Pos_{Max}, we abstract this state to “train has passed the crossing”.

The speed of the train is assumed to be constant, unless an emergency break is signaled. This is modeled as follows:

$$\begin{aligned} Speed &: 0..Speed_{Max} \\ Speed_{t=0} &:= Speed_{Max} \\ Speed_{t=n+1} &:= \begin{cases} Speed_{t=n} - dec_{t=n}, & if \; Speed_{t=n} - Dec_{t=n} \geq 0 \\ 0, & otherwise \end{cases} \end{aligned}$$

For the case study only deceleration is analyzed. The model can however easily extended to acceleration as well. Deceleration is controlled by *TrainControl*. It is by default 0 unless *TrainControl* is in state *brake*. *Error_brake* may prevent braking.

$$\begin{aligned} Dec &: 0..Dec_{Max} \\ Dec_{t=0} &:= 0 \\ Dec_{t=n+1} &:= \begin{cases} Dec_{Max}, & if \; TrainControl = brake \wedge \neg error\_actuator \\ 0, & otherwise \end{cases} \end{aligned}$$

4.2 DCCA

This model was used to analyze the system with DCCA as described in Sect. 2. All proofs were done using the SMV model checker tool. The proofs took less than 1 minute (for $Pos_{Max} = 1000$ and $Speed_{Max} = 16$).

First we proved that the system is functionally correct. We showed that the empty set of failure modes is not critical. The next step was to examine the singleton sets. We found that $\{error\_passed\}$ and $\{error\_odo\}$ were the only critical sets. Because the system is functional correct, these two are also minimal critical. To find minimal critical sets with two elements, we had to check only those sets, which do not include $\{error\_passed\}$ or $\{error\_odo\}$. So 6 proofs of criticality were needed. Four of the examined sets them were found to be critical. No more sets of failure modes existed, which not already included on of the minimal critical sets. Altogether DCCA yielded the following complete list of minimal critical sets:

– {error_passed}
– {error_odo}[1]
– {error_comm, error_close}
– {error_comm, error_brake}
– {error_close, error_actuator}
– {error_brake, error_actuator}

This example shows that the effort for a complete DCCA does not grow exponentially in real applications, if monotony is used and an adequate methodology is used. This can be quickly computed with set algorithms. In conclusion, by use of monotony instead of 2^6 proof obligations only 13 proofs were necessary to determine all minimal, critical sets.

The results were very surprising for us. We already did a formal FTA with the semantics of [12] for this system using the interactive theorem prover KIV [1]. The fault tree we have proven to be complete consisted of only OR-gates. This means all leaves of the fault tree are single point of failures. DCCA now shows that only *error_passed* and *error_odo* are single points of failure. Other failure modes, which seem to be very safety critical - like for e.g *error_brake* - are only critical in conjunction with other failures. But this result is also intuitively correct. For example if only the brakes fail and everything else works correctly, then the crossing will be secured in time and there will be no need for an emergency stop at all. This means failure of the brakes is NOT a single-point-of-failure for this system. So this example is a proof of concept that the results of DCCA are not only in theory more precise than formal FTA but also in practice.

5 Related Work

There exist some other methods of formally verifying dependencies between component failures and system failure modes. One such technique is formal FTA [12]. Formal FTA requires, that all inner nodes of a fault tree are formalized. This can be very time consuming and difficult (see the example in Sect. 4). A second problem with formal FTA is, that it relies on universal theorems. But, proof obligations for gates must be universal, since only universal properties can transitively lead to properties for the whole system. DCCA uses existential proof obligations. This allows to distinguish whether an (failure) event is a necessary or sufficient condition.

[1] This failure mode is only critical, if the safety margin in the calculation of *pos_closed_reached* is to small.

Another related approach has been developed in the ESACS project [2]. Here again model checking and FTA is used as basis. The ESACS approach does not require inner nodes of the fault tree to be formalized. However, the approach requires to adjust the model for different proofs. This can be time consuming (building BDDs for model checking is expensive) and

6 Conclusion

We presented a general formal safety analysis technique: DCCA. DCCA is a generalization of the most widely spread safety analysis techniques: FMEA and FTA. In the formal world, verification of functional correctness, formal FMEA and formal FTA may be found as special cases of DCCA. So DCCA may be used to verify different types of safety analysis techniques in a standardized way. The proof obligations of DCCA may be constructed automatically and the proofs can be done - for finite state systems - by model checking.

DCCA formalization is strictly more precise, than other formal formal safety analysis techniques like formal FTA. Theoretically, the effort for DCCA grows exponentially. But we have not found this case to happen in real world applications. The costs are more likely to grow linear (for non redundant systems) or polynomial by n (for systems with n-times redundancy), if monotony is used.

We showed the application of DCCA to a real world case study: the reference case study “radio-based railroad crossing” of german research foundations priority program 1064. DCCA has rigorously identified critical sets of failure modes and the results of the analysis were much more precise than what can be achieved with informal or formal FTA.

References

[1] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in LNCS, pages 363–366. Springer-Verlag, 2000.

[2] P. Bieber, C. Castel, and C. Seguin. Combination of fault tree analysis and model checking for safety assessment of complex systems. In *Dependable Computing EDCC-4: 4th European Dependable Computing Conference*, volume 2485 of *LNCS*, pages 19–31, Toulouse, France, 2002. Springer-Verlag.

[3] J. Fragole J. Minarik II J. Railsback Dr. W. Vesley, Dr. Joanne Dugan. *Fault Tree Handbook with Aerospace Applications.* NASA Office of Safety and Mission Assurance, NASA Headquarters, Washington DC 20546, August 2002.

[4] ECSS. Failure modes, effects and criticality analysis (FMECA). In European Cooperation for Space Standardization, editor, *Space Product Assurance*. ESA Publications, 2001.

[5] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.

[6] T. A. Kletz. Hazop and HAZAN notes on the identification and assessment of hazards. Technical report, The Institution of Chemical Engineers, Rugby, England, 1986.

[7] J. Klose and A. Thums. The STATEMATE reference model of the reference case study 'Verkehrsleittechnik'. Technical Report 2002-01, Universität Augsburg, 2002.

[8] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1990.

[9] F. Ortmeier and W. Reif. Failure-sensitive specification: A formal method for finding failure modes. Technical Report 3, Institut für Informatik, Universität Augsburg, 2004.

[10] Michael R. Beauregard Robin E. McDermott, Raymond J. Mikulak. *The Basics of FMEA.* Quality Resources, 1996.

[11] G. Schellhorn, A. Thums, and W. Reif. Formal fault tree semantics. In *Proceedings of The Sixth World Conference on Integrated Design & Process Technology*, Pasadena, CA, 2002.

[12] A. Thums. *Formale Fehlerbaumanalyse.* PhD thesis, Universität Augsburg, Augsburg, Germany, 2004. (in German), (to appear).

[13] A. Thums and G. Schellhorn. Model checking FTA. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 739–757. Springer-Verlag, 2003.

Dependability Challenges and Education Perspectives

Henrique Madeira

CISUC - University of Coimbra
3030-199 Coimbra, Portugal
henrique@dei.uc.pt
http://www.cisuc.uc.pt

Our society is increasingly dependent on computer systems that play a vital role in practically all aspects of our economy and daily lives. The question about the limits of computer dependability and about the challenges raised by those limits is more acute than ever. These challenges are not merely technological, as the complex facets of the dependability equation also touch methodological aspects, attitudes, and educational issues more and more.

In the past, the terms high dependability, fault tolerance, and safety critical had connotations to high demanding and specific applications such as flight control and nuclear power plants. Today, the networked information society has created an incredible complex pattern of interconnected computer systems, from small embedded systems to very large transactional servers, that interact with users in a huge variety of ways and scenarios. This complex structure of computes, networks, and people comprises the most demanding challenges in computer dependability.

In spite of the significant advances that have been achieved in computer dependability in the past, the situation today requires a resetting of the research effort on computer dependability, in tight connection with other disciplines such as human behavior or sociological sciences. In addition to this interdisciplinary research effort, the educational issues related to computer dependability deserve particular attention, as there is a clear gap between what is known and what is being taught, not only at the system design course syllabuses but also in a more sense of creating a dependability-aware culture.

In the panel we bring to discussion the two complementary aspects mentioned above:

- new challenging problems on computer dependability, as applied to the hardware, software, or human elements, either focusing on specific application domains or addressing future scenarios and visions for the information society and for the computer industry, and
- new ideas on how to promote a dependability-aware culture in all the players/institutions (education, research, industry, research funding, and users), attract new researchers to the field, and highlight the fundamental role of the research on new dependability technologies in the information and communication systems and infrastructures.

Availability in Industry and Science
- A Business Perspective -

Manfred Reitenspieß

Fujitsu Siemens Computers, Otto-Hahn-Ring 6, 81739 München, Germany
manfred.reitenspiess@fujitsu-siemens.com
http://www.fujitsu-siemens.com/products/software/cluster_technology/rtp4.html

Abstract. Despite the continuous stressing of security and safety aspects in IT systems, the deployment of highly available services or secure applications is still low. Very often, the economic aspects for the adoption of new technologies are ignored at their conception. New mobility and mobile services will revolutionize our everyday life – only if they come with adequate quality. Standards are an important means to drive the wide adoption of new technology – if backed by industry and research. A good example for such a combined approach is the Service Availability Forum.

1 Where to Start?

Security and safety have been focus areas of computer science from the beginning and a wide range of methods, technologies and products have been developed and are being developed as of today. Fault tolerant computer systems, certified operating systems, processes for the development of safety critical software and associated tools are available.

With a few notable exceptions (e.g. security gateways or cluster systems) all these technologies have one thing in common: their business potential is limited and they have not been adopted on a broad scale.

A number of reasons may be responsible for this effect: technology, cost, know-how, interest, insufficient ease of use or access and many more. In the rest of this paper, we will look at these issues using mobile services and their availability as an example, and show how these issues can be overcome. Although this does offer a very focused view on the wide area of dependability we think that the following thoughts can be broadened to most if not all aspects of security and safety.

We will use a relatively intuitive understanding of the terms availability and mobility in the rest of the paper. For detailed definitions, the readers are referred to more technical papers such as [1].

2 Availability – The Market and Complexities Are Growing

Although it has been said over and over again - our society's dependency upon Information Technology (IT) is still growing. This trend will continue into the foreseeable future. The effects of this growing dependency are felt in everyday life:

- Chaos rules if baggage cannot be delivered at the airport,
- millions of working hours are lost due to computer viruses,
- billions of dollars are spent to fix bugs in automotive applications.

But: the business potential and the positive effects on life of the increased use of IT for mobile services and for mobility services are far beyond today's imagination:

- ad hoc networks in car to car communication will reduce the time spent in traffic jams,
- remote service using mobile networks will save millions of hours waiting at the car dealer.

At the same time, the technological complexity and the service complexity will increase tremendously. Networks of networks will be formed, speed of communication will multiply, services will be cross connected. New services need to be implemented fast to achieve a short time to market and to create RoI (Return of Investment) quickly.

And there is a high risk that the quality of these services will not be sufficient to be widely adopted. Until now, standards for high-availability components or development environments, though evolving, are not yet widely adopted by the industry. Missing standards means that independent software vendors (ISVs) cannot take advantage of large scale deployments of their high-quality solutions on a number of architectures. Therefore they will only use such tools and methods which do not hinder the easy deployment of their applications.

Without standard availability components and platforms, the implementation of highly available mobile services becomes much more complex and therefore expensive than writing standard quality solutions. And the time to market will increase considerably, binding resources in the development of platforms features. Their proprietary high-availability implementations need to be maintained over the lifetime of a project or product, again making it extremely expensive and taking resources away from the development of value generating service features.

The support for availability features is very limited in standard and widely used development environments such as Java beans or UML. Therefore the integration of high-availability features requires additional efforts, often not spent in the initial phases of a product and extremely difficult to add after a product has been released.

Overall, the know-how about availability and fault-tolerance functions and their implementation is posing a major obstacle to the wide use and deployment of high-quality new services fulfilling the availability requirements of successful, highly scalable mobility and mobile solutions for millions of users.

3 Industry Initiatives Are Under Way, But…

A number of initiatives are ongoing, which address high-availability and reliability aspects in industrial environments. The W3C is specifying web concepts for reliable messaging and transaction support. The PICMG consortium is specifying hardware architectures (Advanced TCA) for the use in so-called carrier-grade environments, i.e. environments showing the same high-availability as traditional telecommunications networks.

The approach, the Service Availability™ Forum (SAForum, www.saforum.org, [2]) has taken, is an excellent example for a focused, but standards based approach to achieve availability levels only known in proprietary telecommunications systems. The initiative was started in 2001 when it became clear that proprietary telecommunications equipments will be more and more replaced by standard IT components. A whole new ecosystem of hardware, middleware and solutions is developed around standards and commercial off-the-shelf IT components. SAForum members are all major IT manufacturers, most major network equipment manufacturers and a number of ISVs. The standards are driven on a marketing level and on a technical level.

The SAForum marketing group promotes the wide adoption of SAForum standards through publications, fair participation, conferences (International Service Availability Symposium), analysts and press briefings. The SAForum technical work group specifies standards for high-availability middleware interfaces, functionality which is in use in legacy telecommunications solutions for a number of years already. The interface categories are:

- HPI (hardware platform interface): specifications for the management of redundant hardware components,
- AIS (application interface specification): interfaces for application developers to create applications with the highest levels of availability,
- SMS (software management specification): interfaces and data structures for the management of redundant, highly available software and hardware structures.

Also on EU level, high-availability is a focus area of research with *IST-2004-2.4.3 - Towards a global dependability and security framework.* This Strategic Objective aims at building technical and scientific excellence, as well as European industrial strength in security, dependability and resilience of systems, services and infrastructures.

Also, the aspects of critical infrastructures have gained a lot of attention recently. The complex interplay between the use of IT and critical infrastructures such as communications networks, finance, air traffic or water supplies are addressed on national and international levels.

4 Integrated Research Is a Must

It should be clear from the above that research needs to take an integrated look at dependability. Of course, a technologically sound basis is indispensable. But to overcome the critical inhibitors of highly dependable mobile services, economic and usability aspects need to be addressed simultaneously.

For each high-availability technology its economic implications need to be discussed. Aspects of adoption, integration with current or new applications, usability are decisive for the success of a new or improved technology.

Key elements of high-availability research need to be:

- Working with standards and industry bodies
- Integration with widely used tools and products
- Dissemination of know-how and expertise
- Economic evaluation of new or updated technologies

- Providing adequate development tools, again with an eye on their integration with widely accepted technologies (Java, UML)
- Supporting the reuse of existing code
- Scalability of solution development as in many cases the acceptance of mobility services is not known in the initial project phases.

The SAForum supports an academic affiliation program to foster a closer cooperation between research and industry. Researchers are invited to participate actively.

References

1. Laprie, Jean-Claude: Dependability - Its Attributes, Impairments and Means. In (B.Randell, J.-C.Laprie, H.Kopetz, B.Littlewood Edts): Predictably Dependable Computing Systems,. Springer, Berlin Heidelberg New York. (1995). ISBN: 3-540-59334-9.
2. Jokiaho, Timo, Penkler, Dave: The Service Availability Forum Platform Interface. TechFocus, December 2002. 53-61

Fast Run-Time Reconfiguration for SEU Injection

David de Andrés, José Albaladejo, Lenin Lemus, and Pedro Gil

Fault Tolerant Systems Research Group,
Universidad Politécnica de Valencia,
DISCA-ETS Informática Aplicada, Campus de Vera #14, E-46021,
Valencia, Spain
{ddandres, jalba, lemus, pgil}@disca.upv.es

Abstract. Mission critical systems are computer-based control systems that people depend on, often for their livelihoods, sometimes for their lives. As the integration scale of VLSI increases, computer-based systems are more prone to be affected by Single Event Upsets (SEUs). FPGAs have proven to be very useful devices for the injection of SEUs in order to assess the dependability of those systems. The conventional approach for injecting SEUs following the Run-Time Reconfiguration methodology involves reconfiguration times dependent on the complexity of the model. In case of complex models, it will greatly increase the execution time of fault injection experiments. This paper presents a new approach for the injection of SEUs into the memory elements of the system under test by means of FPGAs. This approach takes benefit of the FPGA architecture to minimise the reconfiguration time for SEU injection, obtaining an speed-up of near two orders of magnitude.

1 Introduction

In recent years, there has been a great increase in the use of computer-based systems in mission critical systems, such as automotive systems, aircrafts, telecommunication, etc. The occurrence of a failure in this kind of systems can cause the loss of huge amounts of money or even worst the loss of human lives. Therefore, there has been a great interest in developing different techniques to validate the fault tolerance capabilities of these systems and evaluate their dependability.

At the same time, the integration level of electronic circuits is experiencing a continuous increase. The complexity of these new systems is making it difficult to guarantee a certain level of dependability due to the increasing rates of occurrence of transient (also known as *soft errors* or *single event upsets* (SEU)), intermittent and permanent faults [1].

Fault injection appears as a suitable means to evaluate the dependability of this kind of systems. The main aim of fault injection is to enable the analysis of the behaviour of the systems in the presence of faults. The process of introducing such faults into the system in a controlled way is named *injection*. It is well known that the analysis of the results obtained from these experiments is useful to assess the dependability of the system under study.

Fault injection techniques are usually divided into three main groups[2][3]:

- *Hardware Implemented Fault Injection* (HWIFI)[4], which consists in the injection of faults into the real system (or a prototype) by means of *physical* mechanisms.
- *Software Implemented Fault Injection* (SWIFI)[5], which is also applied to the real system (or a prototype), performs the fault injection by means of *logic* mechanisms, such as programs, the operating system, etc.
- *Simulation-Based Fault Injection* (SBFI)[6], which consists in the injection of faults in a *model* of the system.

Although the execution of fault injection experiments by using HWIFI and SWIFI techniques is very fast, these techniques can only be applied on the last stages of the design cycle, since they require the final system or a prototype to be available. Fixing a design fault can be very costly, both in time and money, since it is necessary to iterate again through the design cycle.

On the other hand, SBFI techniques can be applied to a model of the system in the first stages of the design cycle. This allows design faults to be corrected earlier in the design cycle, saving time and money. However, at that time, there is little knowledge about the final implementation of the system. Therefore, it is necessary to develop very detailed and complex models of the system to achieve a desired level of representativeness. Since simulations are performed on a computer system, the time devoted to the simulation of these complex models is very long. Thus, the community is in need of solutions to reduce this simulation time.

The evolution of the FPGAs (Field Programmable Gate Array) architecture and the development of hardware emulators provided a solution for fulfilling the previous requirement by exploiting the reconfigurable capabilities of FPGAs with fault injection purposes [7][8][9]. Reconfiguring the FPGA involves the manipulation of the logic within the device at run-time. The basic idea consists in reconfiguring the device to emulate the behaviour of the system in the presence of faults. This technique has two main advantages: 1) it can be applied in the first stages of the design cycle since it only requires a model of the system and 2) it is executed very fast since it runs on real hardware (there is not any simulation process). In this way, the complex models can be executed on an FPGA at a speed very close to the speed of the final prototype.

Generally, FPGAs lack of the level of observability available in SBFI, but they can be very useful at quickly locating the source of error. After that, an SBFI tool can perform a fault injection focused on these areas previously located by the FPGA-based fault injection.

As previously stated, the appearance of SEU faults into new complex systems is an important issue the community must deal with. The main drawback of FPGA-based fault injection techniques is the need of reconfiguring the system for each fault that must be injected. In fact, the internal architecture of FPGAs makes the injection of SEUs to be a very time consuming process. Thus, it is mandatory to develop new approaches to minimise the time devoted to reconfigure the system for fault injection purposes.

This paper presents a novel approach for the injection of SEUs into the model of a system by means of FPGAs. This approach minimises the amount of bits required for reconfiguring the device and, therefore, greatly decreases the time devoted to the fault injection process. Moreover, the reconfiguration time for this approach is nearly constant and does not depend on the complexity of the system under study.

The structure of this paper is the following. Section 2 presents an overview of the existing FPGA-based fault injection techniques. The conventional and the novel approaches for SEU injection following the run-time reconfiguration methodology are detailed in Section 3. A timing analysis for the SEU injection following each approach is presented in Section 4. Results from practical experimentation are presented in Section 5. Finally, the conclusions and future work can be found in Section 6.

2 FPGA-Based Fault Injection Techniques

Two different methodologies have been developed to inject faults by means of FPGAs. Each one of these methodologies is based on a different approach for the development of reconfigurable applications (see Fig. 1): *compile-time reconfiguration* (CTR) and *run-time reconfiguration* (RTR).

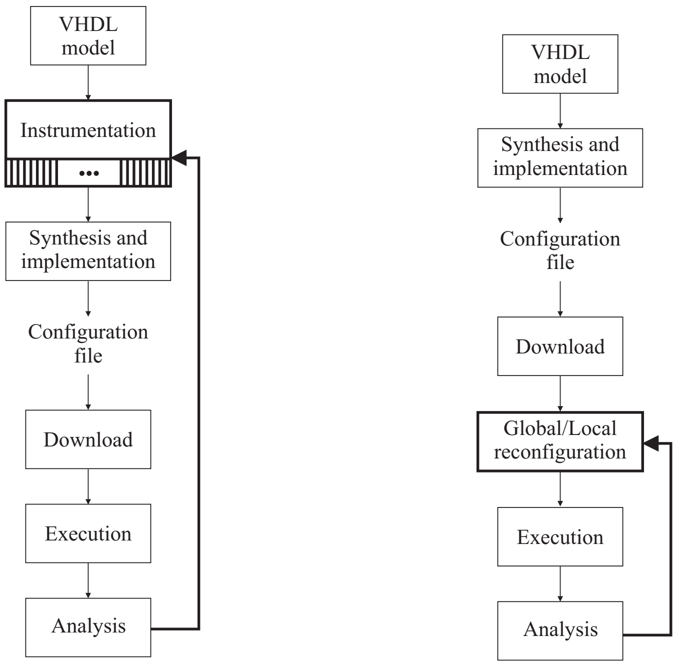


Fig. 1. FPGA-based fault injection methodologies: Compile-Time Reconfiguration (*left*); Run-Time Reconfiguration (*right*)

2.1 Compile-Time Reconfiguration

The CTR technique relies on the instrumentation of the model to make it *injectable* [10][11][12]: the original model is modified to include the logic that emulates the behaviour of the system in the presence of faults. This logic is controlled by the activation of some external signals that are used to perform the fault injection. Usually, this control logic is implemented by means of a mask-chain, which allows the user to perform the fault injection and to get the internal state of the system by using this mask-chain as a shift register.

Obviously, the instrumented model needs more FPGA internal resources than the original model. In the case of complex systems, the instrumentation of the whole model does not fit into the selected device, requiring several partial instrumentations of the model. Since the synthesis and implementation are very timing consuming processes, it seems clear that other solutions must be applied for injecting faults into complex systems.

2.2 Run-Time Reconfiguration

The RTR technique relies on the dynamic and partial reconfiguration capabilities of programmable devices. In that case, reasoning is made in terms of how to reconfigure the internal resources of the device to emulate the behaviour of the system in the presence of faults [13][14][15][16].

The main advantage of this technique is that the original model of the system is not instrumented and, therefore, only one implementation is needed: the fault-free one. Therefore, it does not require any additional resource and appears as a suitable technique for fault injection in very complex systems, which will require several partial instrumentations otherwise.

In this approach, the configuration port of the device is used with fault injection purposes. The reconfiguration of the programmable device, i.e. the fault injection, requires the transfer of a reconfiguration file between the host and the FPGA through this port. Thus, each injection increases the time devoted to the execution of the injection experiment. In the same way, the host can obtain the internal state of the reconfigurable device. Therefore, it is necessary to develop different approaches that minimise the amount of bits to be transferred in order to reconfigure the device for fault injection.

3 Fast Run-Time Reconfiguration for SEU Injection

RTR has been presented as a well suited methodology for the speed-up of SBFI experiments. The fault injection is performed by means of some reconfiguration files that are transferred between the host and the programmable device. In order to minimise the injection time, the amount of bits to be transferred must be kept to a minimum.

The size of the reconfiguration files when using the conventional approach for SEUs injection is dependent on the number of flip-flops (FF) in the system. Therefore, as the models become more complex, these files become larger and the injection process takes longer.

This paper presents a novel approach for the fast injection of SEUs following the RTR methodology. This new approach greatly reduces the time devoted to the fault injection process since the size of the reconfiguration files has been minimised to a large extent. Moreover, the size of these files is no longer dependent on the complexity of the model, making this approach very suitable for any kind of system.

In order to show the mechanisms that are followed by the conventional and the proposed approach in order to perform a SEU injection, it is necessary to provide a model of a basic configurable block for a generic FPGA. The basic configurable block of the main FPGAs in current use, such as Virtex-II [17] from Xilinx and Stratix II [18] from Altera, can be modelled as shown in Fig. 2.

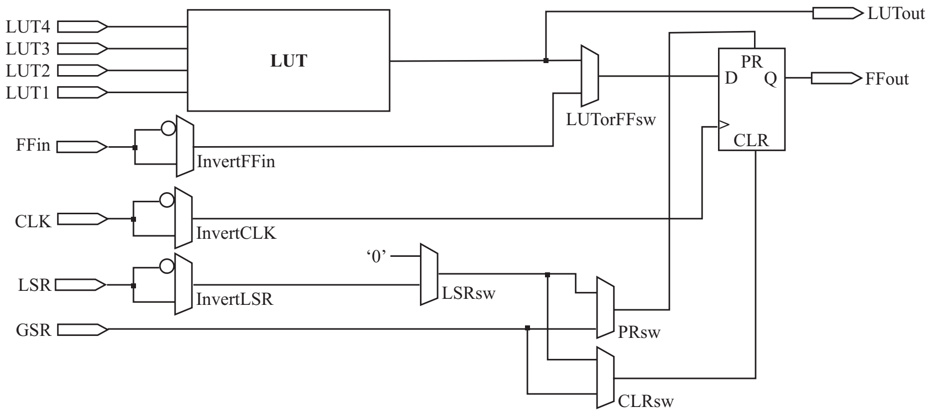


Fig. 2. Schematic view of a basic configurable block of a generic FPGA

As can be seen on Fig. 2, the FF is fed directly from the FFin input or from a *Look-Up Table* (LUT) that implements some combinational logic. It is possible to asynchronously apply a set or reset to the FF by means of the *Global Set Reset* (GSR) and the *Local Set Reset* (LSR) lines. The GSR line drives all the FF of the FPGA and can pulsed via the configuration port of the device. The LSR line only drives one FF and it is mainly used to implement the clear signal for components such as counters. This line can not be externally pulsed. The logic value being driven onto the signals can be inverted by means of the suitable switch.

This model will be used in the following points to present the conventional and the proposed approach for the injection of SEUs following the RTR methodology.

3.1 Conventional Approach

The injection of SEUs involves that the state of a memory element changes to the opposite. The approach proposed in [13][14] focuses on SEUs in the flip-flops (FFs) of the FPGA used as functional memory elements.

As seen in Fig. 2, the only way of changing asynchronously the state of a FF is to apply a set or reset to it. Since it is possible to pulse the GSR line, it can be used to asynchronously change the state of the FF. Whether a set or a reset will be performed depends on the state of the PRsw and CLRsw switches respectively.

Since pulsing the GSR line causes all the FFs of the FPGA to be set/reset, it is necessary to read the state of these FFs previously. Depending on the state of the FF and the SRsw/CLRsw, these switches must be reconfigured to maintain the state of the FF or to flip it.

The SEU injection under this approach consists of the following steps:

1. Start the execution of the application.
   --- Begin the fault injection process ---
2. Stop the execution. A time trigger is used for fault injection.
3. Read the state of all FFs.
4. Change the state of any required PRsw/CLRsw according to the state of its FF.
5. Pulse the GSR line. This will change the state of the desired FF.
6. Change the state of any previously modified PRsw/CLRsw to its original state.
7. Resume the execution.
   --- End the fault injection process ---
8. End the execution. Record the observations and perform their analysis if required.

It is to note that the SEU injection involves two different operations: 1) read the state of all FFs; 2) reconfigure the state of some switches. The time devoted to both operations is dependent on the number of FFs in the system and their state: more FFs means more information to be transferred between the host and the FPGA.

According to this result, the injection of SEUs in complex systems will greatly increase the execution time of experiments. Thus, the necessity of reducing the reconfiguration time for the injection of SEUs arises.

3.2 Novel Approach

The main problem of the conventional approach is that the GSR line drives the set/reset lines of all the FFs of the device. This leads to large reconfiguration files for complex systems: read the state of all the FFs and reconfigure almost all the related switches.

The generic architecture shown in Fig. 2 presents a LSR line that enables to asynchronously set/reset the state of just one FF. The use of this line would dramatically reduce the size of the reconfiguration files: read the state of just one FF and reconfigure a couple of switches.

As previously explained, the LSR line is mainly used for the implementation of components such as counters. Therefore, there is no means to pulse the LSR line externally since it will be driven by the logic of the design. The evolution of the state of the system will determine when to apply a set/reset to the FF via the LSR line.

However, a deeper look at Fig. 2 will provide solution for this problem. It shows that the LSR line has a switch (InvertLSR) that is in charge of inverting the logic value of the driven signal if required. There is no way to tell whether a high or low logic level is being driven into the line, but both levels can be obtained just by changing the state of InvertLSR.

Let us assume that, after reconfiguring PRsw/CLRsw properly and activating the LSRsw switch, a low logic level is being driven into the LSR line and the InvertLSR switch is not inverting this value. Changing the state of InvertLSR will drive a high logic level into the line and, therefore, will set/reset the FF. In case the InvertLSR switch was already inverting the logic level of the line, the FF will already be set/reset before changing the state of InvertLSR and, therefore, the goal will already be achieved. The same reasoning can be applied to the case when a high logic level is being applied into the line. Thus, this approach assures that the FF will change its state.

The steps this approach involves are:

1. Start the execution of the application.
   --- Begin the fault injection process ---
2. Stop the execution. A time trigger is used for fault injection.
3. Read the state of the FF that is to be affected by the SEU.
4. Change the state of the PRsw/CLRsw according to the state of its FF.
5. Change the state of the LSRsw to enable the LSR line to set/reset the FF.
6. Change the state of the InvertLSR to assure that the FF will be set/reset.
7. Change the state of the InvertLSR to its previous value.
8. Change the state of the LSRsw to disable the LSR line to set/reset the FF.
9. Change the state of the PRsw/CLRsw previously modified to their original state.
10. Resume the execution.
    --- End the fault injection process ---
11. End the execution. Record the observations and perform their analysis if required.

This approach involves the transfer of fewer bits than the conventional one. However, it assumes that the logic value of the LSR line is being driven by the logic. It could seem that in case the LSR line is not being used for a particular FF this approach could not be used. This special case that, instead of being a problem, is a faster approach is presented on the following section.

Special Case. The proposed approach relies on the LSR line to be driven by the logic of the design. If this line is not routed in the design, no logic value will be driven into the line and, therefore, this approach will not be feasible.

However, it is to note that the unrouted lines, i.e. those that are not being used by the design, are not left as *open* lines. Usually, to avoid any kind of problem arising from this fact, all the unrouted lines are bound to a high logic level. According to this, just the simple activation of the LSRsw will allow the LSR line, if unrouted, to change the state of the FF.

This special case reduces the number of steps required to inject a SEU:

1. Start the execution of the application.
   --- Begin the fault injection process ---
2. Stop the execution. A time trigger is used for fault injection.
3. Read the state of the FF that is to be affected by the SEU.
4. Change the state of the PRsw/CLRsw according to the state of its FF.
5. Change the state of the LSRsw to enable the set/reset of the FF.
6. Change the state of the LSRsw to its previous value.
7. Change the state of the PRsw/CLRsw previously modified to their original state.
8. Resume the execution.
   --- End the fault injection process ---
9. End the execution. Record the observations and perform their analysis if required.

This is the simplest possible approach: the LSR line is not being used so it is bounded to a high logic level. It greatly reduces the amount of bits to be transferred in order to reconfigure the device.

4 Theoretical Analysis

At first sight, it seems that this novel approach is faster than the conventional one. However, in order to validate the feasibility of this approach, an estimation of the time devoted to the SEU injection has been carried out.

Since this approach relies on the RTR capabilities of the device, the platform that has been selected to carry out the experiments is a prototype board from Celoxica [19] (RC1000PP with a Virtex XCV1000 FPGA from Xilinx). The *SelectMap* interface of Virtex FPGAs [20] enables the partial reconfiguration of the device [21] and the read-back of its internal state. The minimum reconfiguration unit is called a *frame* and it consists of 1248 bits for a XCV1000 FPGA. This device contains 96 columns and 64 rows of configurable logic blocks (see Fig. 3) that hold 4 FFs each, for a total of 24576 FFs.

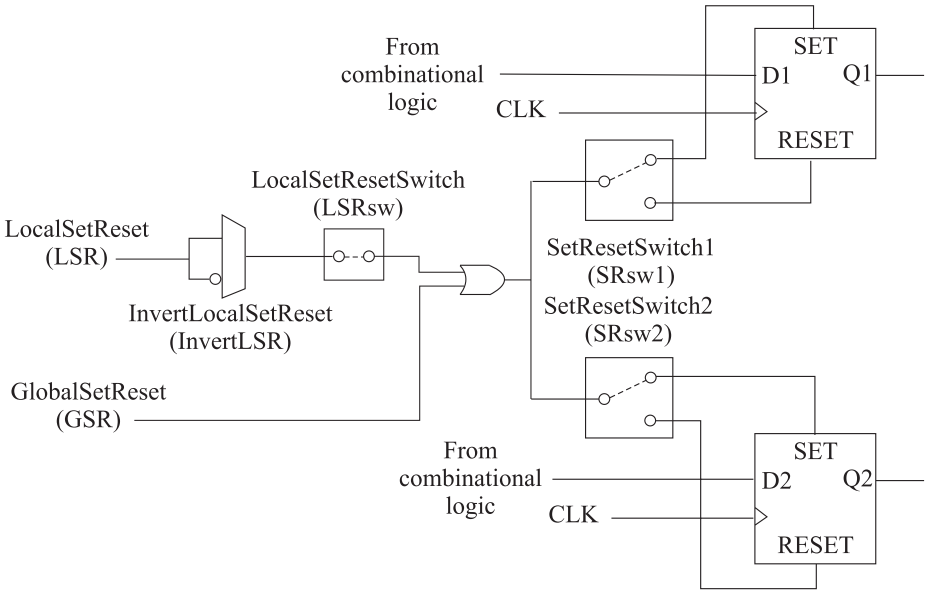


Fig. 3. Schematic view of a basic configurable block of Virtex architecture

A software package named *JBits* [22] has been used to make easier the modification of the configuration file and the RTR of the FPGA. This Java API allows the user to develop applications that are able to analyse the configuration file of the FPGA and to read/write contents from/to the configuration memory of the device. A typical *Readback* command (see Table 1) consists of seven 32-bit words (224 bits) and obtaining the state of one FF involves the transfer of two frames (2496 bits).

Table 1. Typical Readback command of Virtex FPGA

AA995566	;Synchronisation Word
30002001	;Packet Header: Write to Frame Address
00B40400	;Packet Data: CLB, MJA = 90, MNA = 2 (Location to read from)
30008001	;Packet Header: Write to Command
00000004	;Packet Data: Read Configuration Data
2800604E	;Packet Header: Read Frame Data Output (Word Count=2 frames)
00000000	;Dummy Word

The selected prototype board is connected to a PCI slot of the host computer. The PCI interface allows for transfer rates up to 33 MHz and the *SelectMap* interface of Virtex FPGAs can transfer 8 bits in parallel. Thus, the maximum bandwidth between the host and the board is of 264000 bits per second.

The next sections detail the timing analysis for the injection of SEUs following the conventional and the novel approaches.

4.1 Conventional Approach

This approach relies on the use of the GSR line to asynchronously change the state of the selected FF. Since this is a global line for all the FFs of the device, it is necessary to readback the state of these FFs. In the worst case, the state of all the FFs of the FPGA must be read. The total number of bits involved in this operation is carried out in Equation 1:

$$\begin{gathered}\text{Readback all FFs } (\textit{allFF})= \\ (\text{readback 1 FF + readback command}) * \#\text{FFs per column} * \#\text{columns} = \\ (2496 + 224) * 4 * 96 = 1044480 \text{ bits}\ .\end{gathered} \tag{1}$$

Facing once more the worst case, all the SRsw1 and SRsw2 must be reconfigured in order to maintain or invert the state of each FF. The number of bits involved in this operation is carried out by analysing the partial reconfiguration file provided by JBits. This file contains 1022080 bits. The GSR line is automatically pulsed after downloading the reconfiguration file into the FPGA. After that it is necessary to revert the configuration of the device to the original one by downloading an equally sized file.

The time devoted to the injection of one SEU using this approach is obtained by dividing the number of bits to be transferred by the number of bits transferred at a time and the transfer rate. The final result is shown in Equation 2.

$$\begin{gathered}\text{Time to inject one SEU = Number of bits / (transfer rate * bits in parallel) =} \\ (\textit{allFF} + \text{bits to reconfigure all switches} * 2) / (\text{transfer rate} * \text{bits in parallel}) = \\ (1044480 + 1022080 * 2) / (33000 * 8) = 11.699 \text{ milliseconds}\ .\end{gathered} \tag{2}$$

4.2 Novel Approach

This approach assumes that the LSR line can be used to change the state of the FF that is driven by this line. Since we are dealing with the Virtex architecture this line controls de set/reset logic of two FFs [21] (see Fig. 3).

Thus, it is necessary to readback the state of only two FFs, not of all the FFs of the FPGA. The number of bits involved in this operation is shown in Equation 3.

$$\begin{gathered}\text{Readback of two FFs } (2FF) = (\text{readback 1 FF} + \text{readback command}) * 2\text{ FFs}= \\ (2496 + 224) * 2 = 5440\text{ bits}\,.\end{gathered} \tag{3}$$

In the worst case four elements must be reconfigured to flip the state of the selected FF and keep the state of the other (SRsw1, SRsw2, LSRsw and InvertLSR). The required reconfiguration file has 10144 bits.

After that, the state of the InvertLSR is restored to assure that the target FF is set/reset. Then, the rest of the elements must be reconfigured to their original state. In the first step, the state of the LSRsw is changed by transferring 4832 bits. In the second phase, the remaining two elements are reconfigured via 7488 bits.

In this way, the time required to inject a SEU in a FF can be seen in Equation 4.

$$\begin{gathered}\text{Time to inject a SEU} = \text{Number of bits} / (\text{transfer rate} * \text{bits in parallel}) = \\ (2FF + \text{bits to reconfigure 4, 1, 1, 2 elements})/(\text{transfer rate} * \text{bits in parallel})= \\ (5440 + 10144 + 4832 + 4832 + 7488) / (33000 * 8) = 0.124\text{ milliseconds}\,.\end{gathered} \tag{4}$$

Special Case. As stated previously, this special case assumes that the LSR line has not been routed (it is not being used by the design). In that case, the injection of a SEU is similar to the previous one, but it is not necessary modify the configuration of the InvertLSR.

Following the same reasoning as in Equation 4, the time required to inject a SEU in this special case is presented in Equation 5.

$$\begin{gathered}\text{Time to inject a SEU} = \text{Number of bits} / (\text{transfer rate} * \text{bits in parallel}) = \\ (2FF + \text{bits to reconfigure 3, 1, 2 elements}) / (\text{transfer rate} * \text{bits in parallel}) = \\ (5440 + 10144 + 4832 + 7488) / (33000 * 8) = 0.10569\text{ milliseconds}\,.\end{gathered} \tag{5}$$

4.3 Comparison

As it can be seen in Table 2, the two proposed approaches greatly speed-up the SEU injection process compared to the conventional approach for the RTR methodology. The theoretical speed-up obtained is about two orders of magnitude.

Table 2. Estimated comparison of injection times in the worst case

Run-time reconfiguration	Time to inject a SEU	Speed-up
Conventional approach	11.699 ms	-
Novel approach	0.124 ms	94
Novel approach: special case	0.10569 ms	110

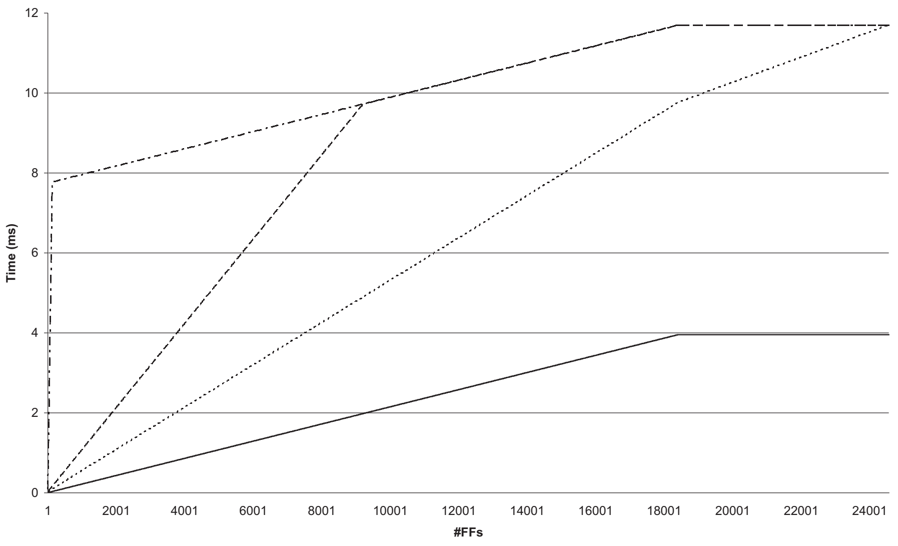


Fig. 4. Estimated injection times for the conventional approach depending on the number of FFs of the system. The best case (*solid line*) assumes that all the switches are properly configured and the SEU injection can be performed without reconfiguring any of them. The worst case (*dashed-dotted line*) takes place when the FFs are spread throughout the FPGA taking as many frames as they can and all the switches must be reconfigured. An optimistic case (*dotted line*) assumes that, although the switches must be reconfigured, they all take the same frames and therefore it minimises the number of frames to be downloaded. On the other hand, a more realistic case (*dashed line*) takes into account an estimation of the distribution of the FFs and switches configuration

Since the worst case is quite extreme, Fig. 4 shows an estimation of the time devoted to the SEU injection for designs ranging from 1 to 24576 FFs.

The same reasoning can be followed for the novel approach and a comparison against the conventional approach estimation can be found in Fig. 5.

Taking a reasonable estimation, although a little optimistic, the conventional approach can be used with good results up to 73 FFs (it assumes that nearly all the FFs are in the same frame and very few switches must change its configuration). The novel approach takes always the same time and, therefore, can be used with very good results for the SEU injection in complex systems.

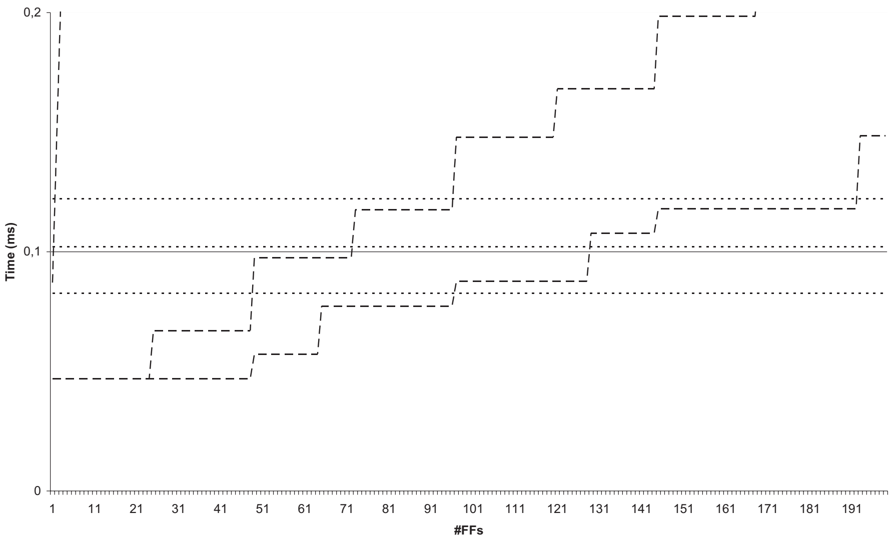


Fig. 5. Comparison between the timing estimation for the conventional (*dashed lines*) and the novel (*dotted lines*) approaches. The three different lines for each of the approaches are, from bottom to top, the optimistic, the more realistic and the worst case respectively

The following section focuses on the implementation of these approaches onto the prototyping board in order to validate the results of the theoretical analysis.

5 Experimental Results

The experimental platform that has been considered is the RC1000PP board from Celoxica [19]. Two systems of different complexity (number of FFs) have been considered to validate the correctness of previous results.

The simplest system consists of an structural model of a PIC16C5X microcontroller [23] running the bubblesort algorithm to sort ten numbers. The implementation of this system takes 287 FFs, occupying the 1% of the configurable logic blocks of the FPGA. In order to increase the complexity of the system under analysis, a different model that includes several replicas of the PIC core has been developed. This little more complex system takes 2967 FFs, occupying the 12% of the available configurable logic blocks.

Several fault injection experiments have been conducted on the two systems: 1000 SEUs have been injected using the conventional approach and another 1000 SEUs have been injected following the novel approach. The FFs affected by the SEU were selected among the FFs used in the design following a uniform distribution. The injection time also follows a uniform distribution along the workload duration. The summary of these campaigns can be found in Table 3 and Table 4.

Table 3. Mean time to inject one SEU in a XCV1000 device for a simple system

	Simple system (1% of total FFs)		
	Conventional approach	Novel approach	Novel approach: special case
#Experiments	1000	935	65
State readback	51.319 ms	1.008 ms	0.927 ms
Reconfiguration	2.146 ms	0.450 ms	0.242 ms
State restoring	2133 ms	0.463 ms	0.435 ms
SEU injection	55.599 ms	1.922 ms	1.606 ms

Table 4. Mean time to inject one SEU in a XCV1000 device for a more complex system

	More complex system (12% of total FFs)		
	Conventional approach	Novel approach	Novel approach: special case
#Experiments	1000	946	55
State readback	170.515 ms	0.977 ms	0.922 ms
Reconfiguration	6.337 ms	0.448 ms	0.236 ms
State restoring	6.2963 ms	0.434 ms	0.433 ms
SEU injection	183.149 ms	1.861 ms	1.593 ms

As it was expected from the previous analysis, the injection time for the conventional approach increases as the number of FFs in the design does. The injection process took a mean of 55.599ms in the simple system and 183.149ms in the more complex one. Nearly the 55% of the execution time was devoted to perform the injection process in this approach.

On the other hand, the SEU injection time for the novel proposal keeps nearly constant regardless of the system complexity. As previously estimated, the injection time decreases when the LSR line is unrouted (special case). However, this special case can only be applied in a reduced number of FFs since the LSR line is usually used by the logic of the system to control the set/reset of the FF. In the system that is being considered, the special case could only be applied to the 6% of the FFs.

Surprisingly, the injection times for both approaches were between 15 and 20 times greater than expected. In [13], this problem was due to both the bandwidth of the interface between the board and the host (parallel port with 50Kbps in read mode) and the configuration clock of the FPGA (4KHz). In our case, the prototyping board has a PCI8090 interface that achieves a throughput of 122MB/s. The problem here may be related to the configuration clock of the FPGA, which is being driven by the application running in the host. The software is only providing a clock rate of near 2 MHz. Furthermore, the PCI bus is an arbitrated bus, which means that the prototyping board must contend with the rest of PCI boards to gain control on the bus. For instance, the mean time for reading back the state of 2 FFs for the novel approach is about

0.997ms. However, there appear eight experiments with readback times ranging from 6.834ms to 11.063ms. It is obviously necessary to improve these issues to obtain better results.

These results show the great speed-up that can be achieved when using the proposed approach to perform SEU injection into the FFs of FPGAs. Not only it is faster method, but it does not depend on the complexity of the system under analysis.

6 Conclusions and Future Work

FPGAs have proven to be very suitable devices for fault injection purposes: these devices provide quick execution times and enable the validation of the system in the first stages of the design cycle.

The two main FPGA-based fault injection methodologies rely on *compile-time* and *run-time reconfiguration* respectively. The former technique requires the model to be instrumented. Complex models may not fit into the FPGA device once instrumented. This leads to the implementation of several partial instrumentations and, therefore, to very long previous phases. The latter technique requires the reconfiguration of the programmable device on the fly. This is the best approach for very complex models (current models of real systems) since it only requires one implementation. Its main drawback is that the reconfiguration of the FPGA increases the execution time of fault injection experiments.

This paper presents a novel approach for the fast injection of SEUs following the RTR methodology. In [13], it was stated that the only way of injecting SEUs with this methodology was by pulsing the line that globally sets/resets all the FFs in the FPGA. The steps involved in this operation greatly increase the execution time of the fault injection experiments and, moreover, this time is proportional to the number of FFs in the design. Therefore, it is necessary to study new approaches for the injection of SEUs.

The approach that has been proposed in this paper makes use of the lines that are in charge of asynchronously setting/resetting the FFs of the system individually. Following this approach, the amount of information that must be transferred between the host and the programmable device is greatly reduced. The speed-up that can be achieved in the fault injection process is, theoretically, of nearly two orders of magnitude. What is more, the time devoted to the injection process is not dependent on the complexity of the system.

Our future work will focus on the following issues:

- New complex systems will require applying not only SEUs, but also some other fault models such as *stuck-at*, *bridging*, *short*, *open*, etc, and even *multiple faults*. How these fault models can be emulated by means of FPGAs is an important issue.
- The amount of bits required to reconfigure the device must be minimise in all cases. The analysis of other approaches must be addressed to decrease as much as possible the time devoted to the fault injection process when applying the RTR methodology.
- RTR not only speeds-up SBFI experiments but it can also be used to evaluate the dependability of FPGA-based systems. Nowadays, more and more systems are being implemented on programmable devices. The assessment of the dependability of these systems is a very important topic and must be studied in depth.

The improvement of the interface between the board and the host, including higher bandwidth and higher configuration clock rate, is also a subject of further work.

References

1. Gil P et.al: Fault Representativeness. Deliverable ETIE2, Report from the "Dependability Benchmarking" Project (IST-2000-25425). http://www.laas.fr/DBench/index.html, (2002)
2. Iyer R K: Experimental Evaluation. International Symposium on Fault-Tolerant Computing – Special Issue. Pasadena, California, USA (1995), 115-132
3. Hsueh M-C, Tsai T K, Iyer R K: Fault injection techniques and tools. IEEE Computer, Vol. 30, nº 4 (1997) 75-82
4. Arlat J, Crouzet Y, Karlsson J, Folkesson P, Fuchs E, Leber G: Comparison of Physical and Software-Implemented Fault Injection Techniques. IEEE Transactions on Computers, Vol. 52, nº 9 (2003) 1115-1133
5. Fabre J C, Sallés F, Rodríguez M, Arlat J: Assessment of COTS microkernel-based system by fault injection. International Working Conference on Dependable Computing for Critical Applications. San José, USA (1999) 19-38
6. Gil D, Gracia J, Baraza J C, Gil P: Study, comparison and application of different VHDL-based fault injection techniques for the experimental validation of a fault tolerant system. Microelectronics Journal, Vol. 34, nº 1 (2003) 41-51
7. Burgun L, Reblewski F, Fenelon G, Barbier J, Lepape O: Serial Fault Emulation. Design Automation Conference (DAC). Las Vegas, USA (1996) 801-806
8. Cheng K-T, Huang S-Y, Dai W-J: Fault Emulation: A New Methodology for Fault Grading. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (CADICS), Vol. 18, nº 10 (1999) 1487-1495
9. Hong J-H, Hwang S-A, Wu C-W: An FPGA-based Hardware Emulator for Fast Fault Emulation. IEEE 39th Midwest Symposium on Circuits and Systems, Vol. 1. Ames, USA (1996) 345-348
10. Hwang S-A, Hong J-H, Wu C-W: Sequential Circuit Fault Simulation Using Logic Emulation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (CADICS), Vol. 17, nº 8 (1998) 724-736
11. Civera P, Macchiarulo L, Rebaudengo M, Sonza Reorda M, Violante M: An FPGA-Based Approach for Speeding-Up Fault Injection Campaigns on Safety-Critical Circuits. Journal of Electronic Testing: Theory and Applications, nº 18 (2002) 261-271
12. Civera P, Macchiarulo L, Rebaudengo M, Sonza Reorda M, Violante M: New Techniques for efficiently assessing reliability of SOCs. Microelectronics Journal, Vol. 34, nº 1 (2003) 53-61
13. Antoni L, Leveugle R, Fehér B: Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes. IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT). Vancouver, Canada (2002) 245-253
14. Antoni L, Leveugle R, Fehér B: Using Run-Time Reconfiguration for Fault Injection Applications. IEEE Transactions on Instrumentation and Measurement, Vol. 52, nº 5 (2003) 1468-1473
15. Parreira A, Teixeira J P, Santos M: A Novel Approach to FPGA-Based Hardware Fault Modeling and Simulation. 6th IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS). Poznan, Poland (2003) 17-24

16. Parreira A, Teixeira J P, Santos M: Built-In Self-Test Preparation in FPGAs. 7th IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS). Tatrmaská Lomnica, Slovak Republic (2004)
17. Stratix II Device Handbook, Volume 1. Altera Corp. SII5v1-2.1. www.altera.com (2005)
18. Virtex-II Platform FPGAs: Complete Data Sheet. Xilinx Corp. DS031, v3.3. www.xilinx.com (2004)
19. RC1000 Functional Reference Manual. Celoxica Inc. RM-1140-0. www.celoxica.com (2001)
20. Virtex 2.5V field programmable gate arrays. Xilinx Corp. DS003-2, v 2.6. www.xilinx.com (2001)
21. Virtex FPGA series configuration and readback. Xilinx Corp. XAPP138, v 2.5. www.xilinx.com(2001)
22. Guccione S, Levi D, Sundararajan P: JBits: A Java-based Interface for Reconfigurable Computing. 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD) (1999)
23. Romani E: Structural PIC165X microcontroller. The Hamburg CHDL Archive. http://tech-www.informatik.uni-hamburg.de/vhdl/ (1998)

Assembly-Level Pre-injection Analysis for Improving Fault Injection Efficiency

Raul Barbosa, Jonny Vinter, Peter Folkesson, and Johan Karlsson

Department of Computer Engineering,
Chalmers University of Technology,
S-412 96 Göteborg, Sweden
{rbarbosa, vinter, peterf, johan}@ce.chalmers.se

Abstract. This paper describes a fully automated pre-injection analysis technique aimed at reducing the cost of fault injection campaigns. The technique optimizes the fault-space by utilizing assembly-level knowledge of the target system in order to place single bit-flips in registers and memory locations only immediately before these are read by the executed instructions. This way, faults (time-location pairs) that are overwritten or have identical impact on program execution are removed. Experimental results obtained by random sampling of the optimized fault-space and the complete (non-optimized) fault-space are compared for two different workloads running on a MPC565 microcontroller. The pre-injection analysis yields an increase of one order of magnitude in the effectiveness of faults, a reduction of the fault-space of two orders of magnitude in the case of CPU-registers and four to five orders of magnitude in the case of memory locations, while preserving a similar estimation of the error detection coverage.

1 Introduction

Computer systems are increasingly being used in safety-critical applications such as aerospace or vehicular systems. To achieve the high safety integrity levels required by these applications, systems are designed with fault tolerance mechanisms in order to deliver correct service even in the presence of faults. Faults may, for instance, occur when processors are disturbed by high energy particles such as neutrons or heavy-ions. Such particles may sometimes interfere with the processor and cause a single event upset (SEU) – an error that typically changes the state of a single bit in the system.

In order to validate the correctness and efficiency of their fault tolerance features, safety-critical systems must be thoroughly tested. Fault injection has become an effective technique for the experimental dependability validation of computer systems. The objective of fault injection is to test fault tolerance mechanisms and measure system dependability by introducing artificial faults and errors.

A problem commonly observed during fault injection campaigns is that not all faults fulfil the purpose of disturbing the system [1]. Often 80-90% of randomly injected faults are not activated [1, 2]. A fault placed in a register just before the register is written or faults that are injected into unused memory locations are examples of faults with no possibility of activation. In most tools the location and the time for fault injection are chosen randomly from the complete fault-space, which is typically extremely large. The statistical implication of this is that the cost of obtaining appropriate confidence levels of the dependability measures becomes unnecessarily high.

To deal with this and similar problems and to reduce the cost of validation through fault injection, two main classes of analysis techniques have been proposed: pre-injection and post-injection analysis [3]. Post-injection analysis aims at predicting dependability measures using the results of fault injection experiments. Pre-injection analysis, in its turn, uses knowledge of program flow and resource usage to choose the location and time where faults should be injected, before any experiment is performed mean.

This paper presents a pre-injection analysis technique that is applicable to injection of transient bit-flips into CPU user registers and memory locations. The bit-flip fault model is often used in fault injection experiments to emulate the effects of single event upsets and other transient disturbances usual.

The objective of the pre-injection analysis is to optimize[1] the fault-space from which the injected faults are sampled. The analysis uses program execution information to (i) eliminate faults that have no possibility of activation and (ii) find equivalence classes among faults and insert only one of these into the optimized fault-space. This is achieved by applying the following rule: *faults should only be placed in resources immediately before these are read by each instruction.* A bit-flip in any resource[2] will only manifest itself once this resource is read to perform an operation. Delaying the injection of the fault until the moment just before the targeted resource is read accomplishes the two objectives stated above. It should be noted that collapsing all faults in a given class into a single fault in the optimized fault-space may cause a bias in the estimated dependability measures (e.g. error detection coverage). One of the objectives of this research is therefore to investigate the magnitude of this bias.

The pre-injection analysis technique was implemented in the GOOFI (Generic Object-Oriented Fault Injection) [4] tool, for Nexus-based fault injection [2, 5, 6], and is also suitable for implementation in other platforms. The effectiveness of the technique was assessed by comparing fault injection results with results obtained by non-optimized fault injection on the same target system. The system is based on the Motorola MPC565 [7] – a microcontroller aimed at the automotive and other control-intensive applications based on the PowerPC architecture. By applying assembly-level knowledge of this architecture it is possible to identify which resources are read by each executed instruction. This information, along with the time of the fault injections, is used to define the optimized fault-space, which is stored in a database. The fault injection experiments are then conducted by random sampling of faults from the optimized fault-space.

[1] The word *optimize* should not suggest that the optimal fault-space is found but rather an improvement on the usual random approach. Further optimization is therefore achievable.

[2] In this paper we use the word *resource* as a common term for CPU-register, main memory locations and otherstate-elements where bit-flips may occur.

2 Related Research

The resources available in computers are, usually, greater than the needs of the applications executed. This fact motivates a first optimization by injecting faults only in used resources. P. Yuste et al. [2] take, in their experiments, special care to avoid placing faults in empty (i.e. not used) memory regions. They obtained 12% of effective faults and pointed out that a random sampling from an unrestricted fault-space consisting of all possible fault locations (bits) and all time points is not a time-effective approach.

Avoiding unused memory regions might be done manually by analyzing the memory map of the application and choosing the segments (stack, heap, etc.) as valid locations for fault injection. This approach is quite simple but does not consider the dynamical usage of resources along the time dimension.

Studies conducted in the past have shown that error manifestation (rate and effects) is affected by workload [8, 9, 10]. In [11] the concept of failure acceleration was introduced by R. Chillarege and N. Bowen. They achieve fault acceleration by injecting faults only on pages that are currently in use and by using a workload pushing towards the limits in CPU and I/O capacity.

J. Güthoff and V. Sieh presented in [12] the operational-profile-based fault injection. They state that the number of fault injections into a specific system component should be proportional to its utilization. Register utilization is defined as the measure of the probability that an injected fault manifests itself as an error. Additionally, the times for fault injection are selected based on the data life-cycles. A data life-cycle starts with the initialization of a register (write access) and ends with the last read access before the next write access. Under the single bit-flip fault model, faults need to be injected only within the data life-cycles, just before each read access.

A. Benso et al. presented in [13] a set of rules with the purpose of collapsing fault-lists. The rules reduce the fault-list without affecting the accuracy of the results of fault injection campaigns by avoiding the injection of faults for which the behavior can be foreseen.

In [14] T. Tsai et al. introduced a technique named path-based injection. With this technique a fault is injected into a resource that will be used by the test program, given a particular input set. After the manual derivation of the input sets, the path of execution is described in terms of a list of executed basic blocks. For each path, faults are only injected in the utilized resources.

Working in fault injection for the test of fault-tolerant circuits, using VHDL models, a set of techniques for speeding up campaigns is described by L. Berrojo et al. in [15]. One of these techniques is workload dependent fault collapsing. During the reference run (a fault-free execution in order to store the program's normal behavior) all read and write operations on memory elements are tracked with bit granularity. Having this log of read and write operations on each bit of each signal, at the circuit level, all possible bit-flips are then collapsed by (i) marking as silent all bit-flips between an operation (either read or write) and a write operation, and (ii) marking as equivalent all bit-flips between an operation (either read or write) and the subsequent read operation.

J. Arlat et al. [16] increased the efficiency of their fault injection experiments targeting the code segment by logging the control flow activated by the workload processes. If the randomly selected address for fault injection is not part of the log (instruction trace), then the corresponding experiment can simply be skipped (as the outcome is already known).

3 Fault-Space Optimization Method

For single bit-flip fault injection, we define a fault-space to be a set of time-location pairs that determines where and when the bit-flip is injected. The time is selected from an interval during the execution of the workload selected for the experiment. The time granularity is based on the execution of machine instructions, i.e. bit-flips can only be injected between the execution of two machine instructions. The complete (non-optimized) fault-space consists of all possible time-location pairs.

The fault-space optimization method presented in this paper states that faults should only be placed in a resource immediately before the resource is read by an instruction. The following sections describe the input needed for the analysis, the output created and the optimization procedure.

3.1 Optimization Input

In order to determine the optimized fault-space it is necessary to gather information about the code of the application and the computer system executing it:

- Assembly code of the application;
- The Program Counter (PC) trace over time;
- The effective address of each memory read access;
- The definition of which resources are read by each assembly instruction.

In our experimental setup, the assembly code is textual information obtained by disassembling the executable binaries of the application, processed automatically by the optimization program. The Program Counter trace and the values of the General Purpose Registers are stored during the execution of the reference run. The effective address of each memory read access is calculated with these values. The definitions of which resources are read by each assembly instruction are built into the optimization program. These were obtained from Motorola's RISC CPU Reference Manual [17] and are available in [18].

3.2 Optimization Output

The resulting output (the optimized fault-space) consists of a list of possible locations and times for fault injection. The optimization procedure has been adapted to both one-shot applications and control applications executing in loops. Each element on the optimized fault-space contains the following information:

- Control loop index;
- Breakpoint address;
- Number of breakpoint invocations within the control loop;
- The fault injection location.

The **control loop index** is specific for control applications which execute in cycles. It defines the cycle during which a fault should be injected. For applications that do not execute in loops, the control loop index is always set to one. The **breakpoint address** specifies the breakpoint position inside the control loop and the **number of breakpoint invocations** specifies the number of times this breakpoint should be reached before fault injection.

3.3 Performing the Optimization

Using the Program Counter trace over time, the disassembled code of the application is parsed to obtain the sequence of assembly instructions executed. Each of the instructions is then analyzed in order to determine which resources the instruction reads. The pseudo-code for this procedure is presented in Figure 1.

```
FOREACH pc_value IN program_counter_trace DO
   control_loop_index ← current_control_loop ()
   breakpoint_invocation ← breakpoint_invocations_count (pc_value)
   instruction ← instruction_at_code_address (pc_value)
   instruction_read_list ← resources_read_by_instruction (instruction)
   FOREACH resource IN instruction_read_list DO
      useful_fault ← [control_loop_index, pc_value, breakpoint_invocation, resource]
      store_in_database (useful_fault)
   ENDFOREACH
ENDFOREACH
```

Fig. 1. Pseudo-code for the optimization procedure

The most important stage (shown in bold in the pseudo-code) is the identification of the resources read by each instruction. To accomplish this, the first step is to find the definition on the list matching the given instruction. This is done by matching the *opcode* and the *operands*. Then, by examining the possible assembly constructs, the symbols available in the *read list* of the definition are replaced by the resources actually read by the given instruction. Figure 2 illustrates this process.

The instruction at address 39DE8 adds R10 to R11 and stores the result in R5. The definition for this instruction is found in the table and the read list contains rA and rB, respectively, R10 and R11. Since these are the two resources read by this instruction, two new lines are inserted into the fault locations for code address 39DE8 (the control loop index and the breakpoint invocation are assumed to hold the specified values).

The second instruction, at address 39DEC, fetches the memory word addressed by the effective address (R6) + 24 and stores it in R7. Its definition in the table specifies rA and MEM32(d+rA), respectively, R6 and the 32-bit word at 1000+24, as being read. The value 1000 of R6 is obtained during the reference run. The two resources along with the timings are then inserted into the fault-space.

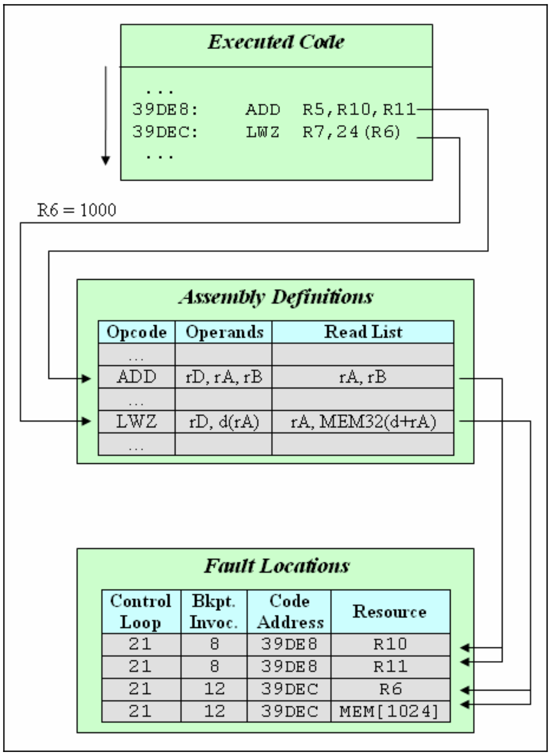


Fig. 2. Example of the optimization procedure

4 Experimental Setup

Figure 3 describes the evaluation platform used to evaluate the effectiveness of the optimization technique for experiments performed on the jet engine control software, which is one of two workloads investigated in this paper. The GOOFI fault injection

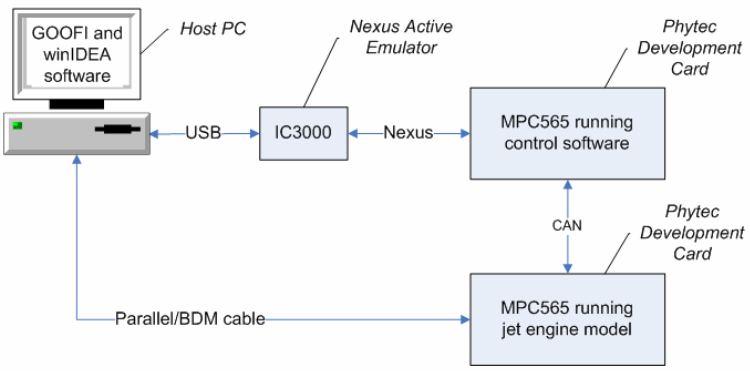


Fig. 3. Evaluation platform for the jet engine application

tool controls the experiments by using the winIDEA debugging environment [19] in conjunction with the iC3000 debugger. Faults are injected into the MPC565 microcontroller running the control software. In the case of the jet engine controller one computer board was used to run the jet engine control software and one board to execute the model of the jet engine. The experimental setup used for the other workload (an implementation of the quicksort algorithm) used only one computer board.

4.1 Fault Injection Tool

GOOFI is a fault injection tool developed at the Department of Computer Engineering, Chalmers University of Technology. It provides the ability to define and conduct fault injection campaigns on a variety of microprocessors. During each campaign GOOFI is responsible for controlling all the necessary software and hardware, and storing the acquired data into a database.

A plug-in [6] has recently been developed in GOOFI which uses the Nexus [5] port to inject faults on Motorola's MPC565. Nexus is an attempt to create a standard on-chip debug interface for embedded applications. This standard is suitable to be used for fault injection [2] since it provides read/write access to the processor's resources and code execution trace capture.

The pre-injection analysis technique was implemented to enhance the existing Nexus fault injection plug-in. The target platform for the current implementation is therefore the MPC565 microcontroller. The technique may however be implemented for any microprocessor.

4.2 MPC565 Microcontroller

The MPC565 is a microcontroller developed by Motorola that implements the PowerPC instruction standard architecture. It is aimed at the high performance automotive market as well as other control-intensive applications. The complete computer system was based on the phyCORE-MPC565 [20] development board. It includes a 32-bit Motorola MPC565 processor, which offers a Nexus debug port enabling real-time trace of program and data flow.

To establish a connection through this port the iSYSTEM iC3000 Active Emulator [21, 22] was used to access the Nexus working environment. The iC3000 emulator was, in its turn, controlled by GOOFI via winIDEA – an integrated development environment offered by iSYSTEM. GOOFI and winIDEA are executing on the same host PC.

4.3 Workloads

Fault injection campaigns were conducted to evaluate the optimization technique using two different workloads: a sort program using the quicksort algorithm and a jet engine controller. Different campaigns targeting registers and data memory, using both optimized and non-optimized fault selection, were carried out. The technique is fully implemented in the sense that all the assembly instructions executed by the workloads are analysed and all registers and data memory locations where optimization is achievable with this method are considered. The outcome of each fault injection experiment was classified into one of the following categories:

- **Detected Error** – All effective errors that are signalled by hardware error detection mechanisms included in the processor.
- **Wrong Output** – All effective errors that are not detected by the processor but lead to the production of wrong results.
- **Non-effective Error** – Errors that do not affect the system execution during the chosen experiment time frame.

4.3.1 Quicksort

The quicksort workload is a recursive implementation of the well-known sorting algorithm. It sorts an array containing seven double-precision floats.

The reference run execution takes two minutes during which the processor is being stepped and all the required data is obtained. The optimization procedure takes 20 seconds to complete. Each fault injection experiment takes less than half a minute to perform. During the execution of the reference run for this application, the MPC565 processor executed 34 distinct assembly instructions (opcodes) and a total of 815 instructions.

4.3.2 Jet Engine Controller

This workload is a control application that executes in loops in order to control a jet engine. At the end of each loop the controller has to produce results and exchange information with the engine (sensor values from the engine and actuator commands from the controller). It is significantly more complex than the quicksort program, allowing the fault-space optimization technique to be evaluated using a real-world application.

The execution of the reference run takes almost 12 hours. The optimization procedure takes 10 minutes to complete. Each fault injection experiment is then performed in less than two minutes for the selected configuration (number of control loops and memory locations to be logged).

Forty control loops of execution were logged during each experiment. From these, ten loops (21 to 30) were chosen as possible temporal locations for fault injection (corresponding to 50ms of real-time execution of the controller). During these ten control loops, in the reference run, the MPC565 processor executed 231.097 instructions. A total of 88 different assembly instructions (opcodes) were executed.

4.4 Fault Model and Fault Selection

The fault model applied is the single bit-flip model of the effects of transient faults. The technique assumes this model as the basis for optimization.

The faults in the non-optimized campaigns were chosen using a uniform distribution. In the case of the optimized campaigns the faults are selected randomly from the optimized fault-space itself (the list of temporal and spatial locations for fault injection described in Section 3.2). This implies that the distribution of faults in resources is proportional to the representation of each resource in the optimized fault-space.

Microprocessor registers were selected as spatial locations for fault injection both in the quicksort and in the jet-engine controller campaigns. Memory locations were only targeted using the jet-engine controller. The registers targeted in the non-optimized campaigns are the ones considered by the optimization method and shown in Table 1.

Table 1. Registers targeted for optimization

General Purpose Registers (32 registers of 32 bits)	Condition Register (32 bits)
Floating Point Registers (32 registers of 64 bits)	Integer Exception Register (32 bits)
Link Register (32 bits)	Count Register (32 bits)

These registers constitute the User Instruction Set Architecture (UISA) Register Set. User-level instructions are limited to this register set while supervisor-level instructions have access to other, special purpose registers (SPRs).

Two limitations of winIDEA (the debugging environment) are important to mention. The floating point registers are only allowed to be injected with faults in the least significant 32 bits. These are the least significant bits of the 52-bit mantissa. The Floating Point Status And Control Register (FPSCR), targeted by the optimization, is also not available for fault injection.

The fault injection campaigns in memory targeted the stack, heap and all other read/write and read-only data segments of the controller. A total of 100KB of memory were targeted as spatial locations.

The analysis of faults in the code segment was still not implemented and was therefore not studied. The optimization is easily extendable to support faults in the code segment by targeting, in each instruction, the 32-bit memory contents addressed by the Program Counter. This would be equivalent to the analysis performed in [16] by using the instruction trace.

5 Experimental Results

5.1 Fault Injection in Registers

Table 2 shows the distribution of the outcomes of faults in the fault injection campaigns targeting microprocessor registers for both the quicksort and the jet engine controller workloads. The quicksort campaigns include approximately the same number of experiments. For the non-optimized jet engine controller campaign, a much higher number of experiments had to be performed in order to increase the confidence in the results.

Table 2. Distribution of outcomes of fault injection in registers

Campaign		*# Exp.*	*Non-effective*	*Detected*	*Wrong Output*
Quicksort	*Random*	2739	2603 (95.0%)	83 (3.0%)	53 (2.0%)
	Optimized	2791	1461 (52.3%)	744 (26.7%)	586 (21.0%)
Jet Engine Controller	*Random*	5708	5457 (95.6%)	200 (3.5%)	51 (0.9%)
	Optimized	1559	964 (61.8%)	466 (29.9%)	129 (8.3%)

The percentage of *effective* faults (*detected* or *wrong output*) increases from 5.0% using non-optimized fault selection to 47.7% choosing faults from the optimized fault-space when targeting the quicksort workload. In the jet engine controller this increase is from 4.4% to 38.2%. The improvement in the effectiveness of faults is, therefore, one order of magnitude.

Table 3 shows the estimated error detection coverage obtained in each campaign. We here define error detection coverage as the quotient between the number of detected and the number of effective faults.

Table 3. Error detection coverage estimations (registers)

Campaign		***Estimated error detection coverage (95% confidence)***
Quicksort	*Random*	61.0 ± 8.2%
	Optimized	55.9 ± 2.7%
Jet Engine Controller	*Random*	79.7 ± 5.0%
	Optimized	78.3 ± 3.3%

The values of the error detection coverage estimations are quite similar whether applying non-optimized or optimized fault selection. In the optimized campaigns the faults are only injected in the location that will activate them (at the time that the register is read). Since no weights are applied to reflect the length of the data life-cycle on the outcomes of faults, it could be expected that the error detection coverage would be skewed.

The detected errors were signalled by the exceptions provided in the MPC565 processor. The distribution among these exceptions is presented in Figures 4 and 5 for the quicksort campaigns, and in Figures 6 and 7 for the jet engine controller campaigns.

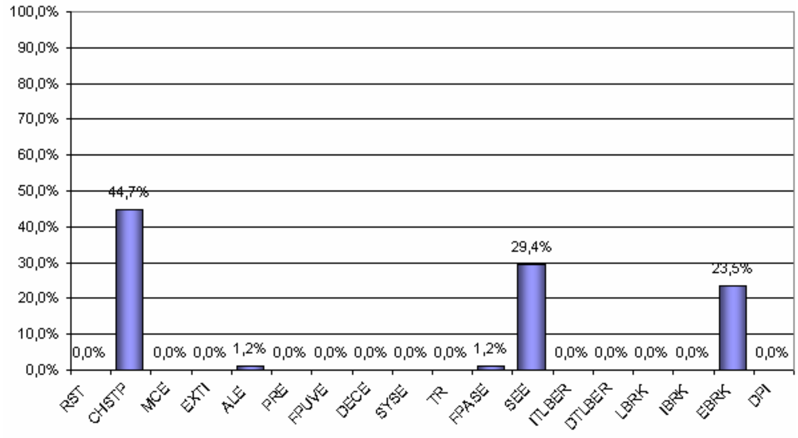


Fig. 4. Exceptions in the quicksort non-optimized campaign (83 faults in registers)

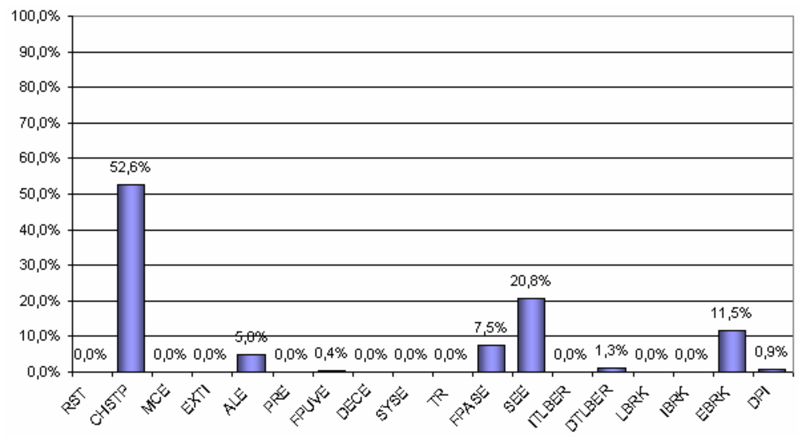


Fig. 5. Exceptions in the quicksort optimized campaign (744 faults in registers)

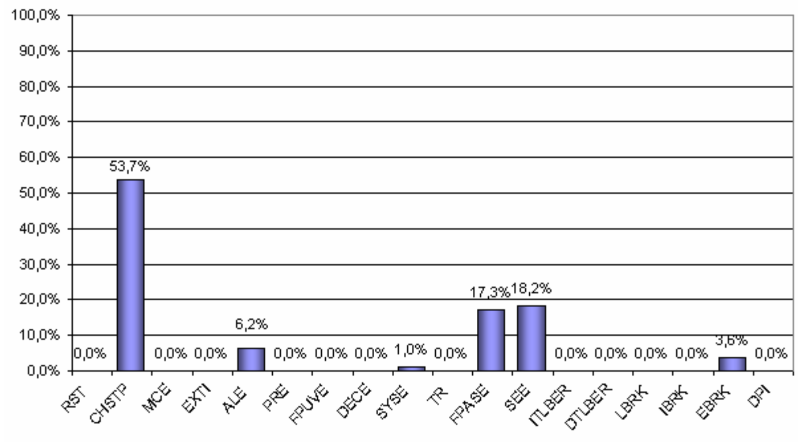


Fig. 6. Exceptions in the jet engine controller non-optimized campaign (200 faults in registers)

It is possible to observe that the detection mechanisms are activated in a similar but not identical way for the non-optimized and the optimized campaigns. Figures 4 to 7 provide an insight on the magnitude of the differences between non-optimized and optimized fault selection. A brief description follows of the most frequently activated exceptions.

Checkstop (CHSTP) – The processor was configured to enter the checkstop state instead of taking the Machine Check Exception (MCE) itself when the MCE occurs. CHSTP does not represent an actual exception, but rather a state of the processor. The processor may also be configured to take the MCE handling routine or enter debug mode. The MCE, which, in this case, leads to the checkstop state, is caused, for instance, when the accessed memory address does not exist.

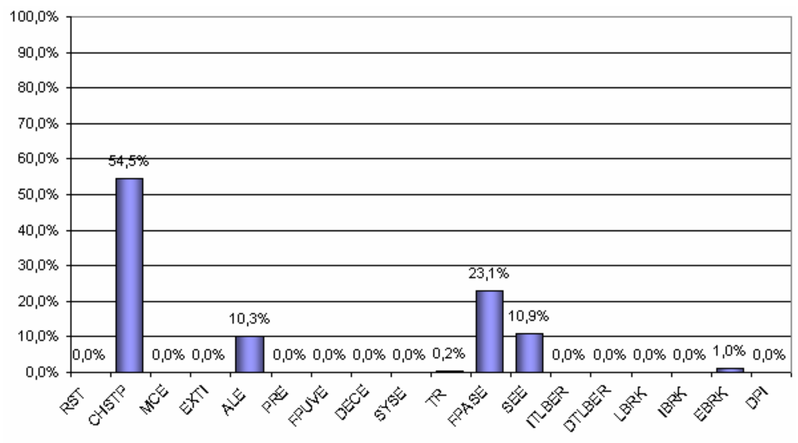


Fig. 7. Exceptions in the jet engine controller optimized campaign (466 faults in registers)

Alignment Exception (ALE) – The alignment exception is triggered under the following conditions:

- The operand of a floating-point load or store instruction is not word-aligned;
- The operand of a load or store multiple instruction is not word-aligned;
- The operand of lwarx or stwcx. is not word-aligned;
- The operand of a load or store instruction is not naturally aligned;
- The processor attempts to execute a multiple or string instruction.

Floating-Point Assist Exception (FPASE) – This exception occurs in the following cases:

- A floating-point enabled exception condition is detected, the corresponding floating-point enable bit in the Floating Point Status And Control Register (FPSCR) is set (exception enabled);
- A tiny result is detected and the floating point underflow exception is disabled;
- In some cases when at least one of the source operands is denormalized.

Software Emulation Exception (SEE) – An implementation-dependent software emulation exception occurs in the following cases:

- An attempt is made to execute an instruction that is not implemented;
- An attempt is made to execute an mtspr or mfspr instruction that specifies an unimplemented Special Puspose Register (SPR).

External Breakpoint Exception (EBRK) – This exception occurs when an external breakpoint is asserted.

Figure 8 shows the distribution of faults per register for the optimized campaign. The figure clearly demonstrates the non-uniform distribution caused by the optimization. The number of faults per register is directly proportional to the number of times the register is read.

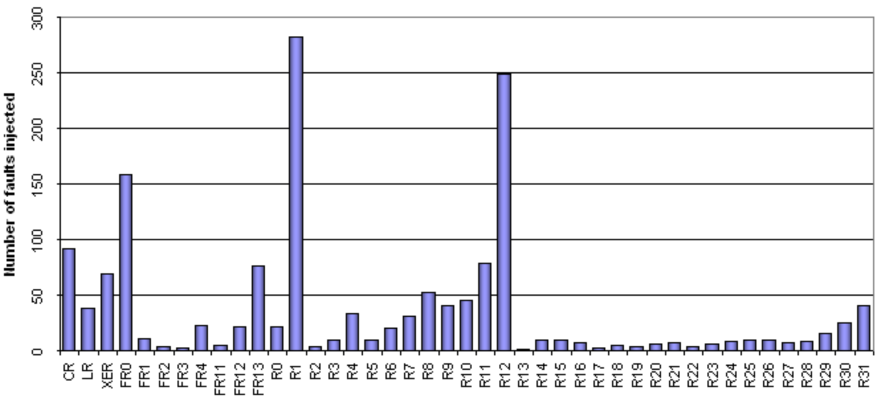


Fig. 8. Number of faults injected per register

5.2 Fault Injection in Memory

Fault injection in memory locations was performed only for the jet engine controller. Table 4 shows the distribution of the outcomes of faults for both non-optimized and optimized fault selection.

Table 4. Distribution of outcomes of fault injection in memory

Campaign		*# Exp.*	*Non-effective*	*Detected*	*Wrong Output*
Jet Engine Controller	*Random*	6666	6532 (98.0%)	40 (0.6%)	94 (1.4%)
	Optimized	2658	2150 (80.9%)	166 (6.3%)	342 (12.8%)

The effectiveness of faults increases from 2.0% using non-optimized fault selection to 19.1% choosing faults from the optimized fault-space. The improvement in the effectiveness of faults is one order of magnitude, similar to one obtained for faults in microprocessor registers.

Table 5 shows the error detection coverage estimations obtained with non-optimized and optimized fault selection.

Table 5. Error detection coverage estimations (memory)

Campaign		*Estimated error detection coverage (95% confidence)*
Jet Engine Controller	*Random*	29.9 ± 7.7%
	Optimized	32.7 ± 4.1%

We here observe a similar pattern to that observed for microprocessor registers, where the error detection coverage estimation using non-optimized or optimized fault selection is quite similar. In this case the estimation from the non-optimized campaign is not very accurate since the 95% confidence interval is still wide due to the small number of effective faults (2%).

Figures 9 and 10 show the distribution of detected errors among the exception mechanisms for the two campaigns.

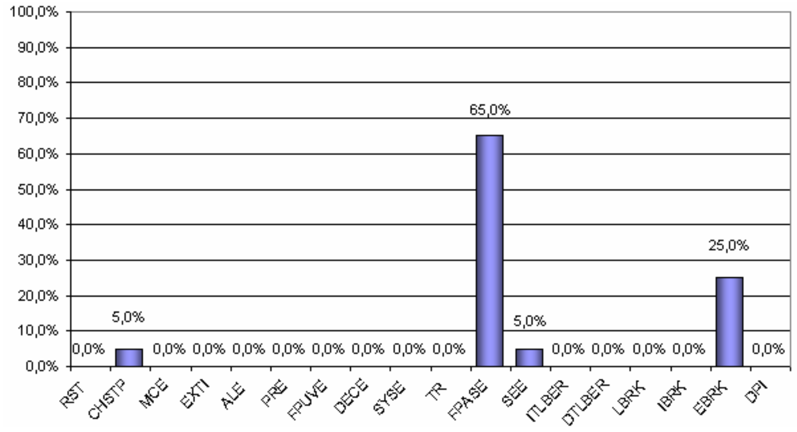


Fig. 9. Exceptions in the jet engine controller non-optimized campaign (40 faults in memory)

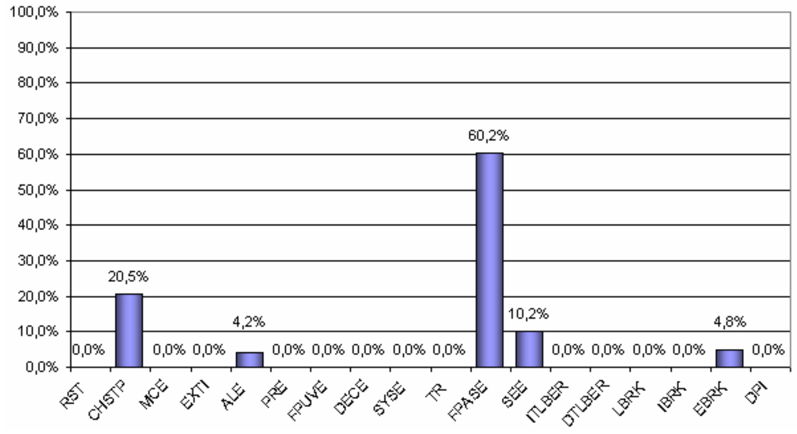


Fig. 10. Exceptions in the jet engine controller optimized campaign (166 faults in memory)

Again, it is possible to observe that the detection mechanisms are activated in a similar but not identical way for the non-optimized and the optimized campaigns.

5.3 Fault-Space Considerations

Applying the optimization method to the fault-space of registers for the jet engine controller resulted in the determination of 7.7×10^6 distinct time-location pairs for bit-flips. All the targeted registers are 32 bit registers[3]. The complete non-optimized fault-space of these registers is obtained by flipping each bit of each register, for each instruction executed. This results in a set containing over 500 million bit-flips. Table 6 summarizes these results.

Table 6. Comparison between fault-space sizes (registers)

Campaign		*Size of the fault-space (time-location pairs for bit-flips)*
Jet Engine Controller	*Random*	5.0×10^8
	Optimized	7.7×10^6
Ratio		1.5%

In the case of the memory fault-space 3.3×10^6 possible time-location pairs for bit-flips were determined using optimized fault selection. The complete fault-space of memory is obtained by flipping each bit of each memory location used by the program, for each instruction executed. Considering a memory usage of 100KB for data by the jet engine controller, the size of the complete fault-space is near 200 billion bit-flips.

Table 7. Comparison between fault-space sizes (memory)

Campaign		*Size of the fault-space (time-location pairs for bit-flips)*
Jet Engine Controller	*Random*	1.9×10^{11}
	Optimized	3.3×10^6
Ratio		0.0017%

6 Conclusions and Future Work

The study presented in this paper shows the efficiency of eliminating faults with no possibility of activation and determining equivalence classes among faults. A comparison with traditional non-optimized fault selection in the complete fault-space shows an order of magnitude increase in the effectiveness of faults. The fault-space itself is reduced two orders of magnitude for the registers and four to five orders of magnitude for the memory. Even though these fault-spaces are still quite large when targeting the complete execution of programs, the exhaustive evaluation of small enough sub-routines against all possible bit-flips becomes possible.

All faults targeting the same bit of a given resource, before this resource is read, are considered equivalent. This way, only one representative of these faults is injected. To obtain an accurate estimation of the error detection coverage (or any other dependability measure) it would be necessary to apply a weight corresponding to the

[3] Floating Point Registers are 64-bits long limited by winIDEA to the least significant 32-bits.

number of faults in each equivalence class. However, the error detection coverage estimated by the optimized fault selection is found to be quite similar to the coverage estimated by non-optimized fault selection.

The analysis of assembly constructors limits the technique to the UISA Register Set. Using a debugger/debugging environment that supports tracing of all read operations on all registers during the reference run would allow the fault-space of all registers to be optimized.

Even though activation of faults is ensured by the optimization technique (activation in the sense that the faulty resources are always utilized) not all faults result in effective errors. This occurs when the data is used in a non-sensitive way by the code (regarding the single bit-flip model). An interesting topic for further studies would be to investigate which activated faults are non-effective and why.

The outcome of a fault is highly dependent on the targeted resource. Faults in some registers were observed to have a greater tendency to cause wrong output while faults in other registers cause detected errors more frequently. This motivates a possible evolution in fault selection by using the results of previous fault injection experiments to select the faults that should be injected next (a combination of pre-injection and post-injection analysis). It would be possible to achieve a faster evaluation of specific error detection mechanisms by injecting faults in the resources that are more likely to activate them.

In the future of fault injection the multiple bit-flip fault model may become more important. Microprocessor technology is employing smaller transistors, with lower power voltages, where a single charged particle is likely to change the state of several bits. It would be appealing to extend the method presented in this paper to improve the selection of multiple bit-flip faults.

A research line orthogonal to the optimization of fault-spaces is the improvement of the path coverage obtained during fault injection campaigns (i.e. consider different control flow decisions and the associated fault-spaces). The presented pre-injection analysis and such a path coverage analysis are complementary and could eventually be combined.

There is still room for further optimization by analyzing the error propagation. When a bit-flip is copied from one resource onto another and the first resource is overwritten, the fault in the new location is equivalent to the fault in the first location. The implementation of an analysis taking advantage of this has been started and preliminary results show additional improvement.

References

1. H. Madeira and J. Silva, "Experimental Evaluation of the Fail-Silent Behavior in Computers Without Error Masking", Proc. FTCS-24, June 1994, pp. 350-359.
2. P. Yuste, J. Ruiz, L. Lemus, P. Gil, "Non-Intrusive Software-Implemented Fault Injection in Embedded Systems", LADC 2003, LNCS 2847, 2003, pp. 23-38.
3. J. Aidemark, P. Folkesson, and J. Karlsson, "Path-Based Error Coverage Prediction", JETTA, Vol. 16, June 2002.
4. J. Aidemark, J. Vinter, P. Folkesson, J. Karlsson, "GOOFI: Generic Object-Oriented Fault Injection Tool", Proc. DSN 2001, July 2001, pp. 83-88.

5. IEEE-ISTO, "The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface", 1999.
6. D. Skarin, J. Vinter, P. Folkesson and J. Karlsson, "Implementation and Usage of the GOOFI MPC565 Nexus Fault Injection Plug-in", Tech. Report No. 04-08, Dept. of Comp. Eng., Chalmers University of Technology, Göteborg, Sweden, 2004.
7. Motorola Inc., "MPC565/MPC566 User's Manual", 2nd edition, 2003.
8. X. Castillo and D. Siewiorek, "Workload, Performance and Reliability of Digital Computer Systems", Proc. FTCS-11, June 1981, pp. 84-89.
9. E. Czeck and D. Siewiorek, "Observations on the Effects of Fault Manifestation as a Function of Workload", IEEE Transactions on Computers, Vol. 41, No. 5, May 1992, pp.559-566.
10. R. Chillarege and R. Iyer, "The Effect of System Workload on Error Latency: An Experimental Study", Proc. ACM SIGMETRICS 1985, August 1985, pp. 69-77.
11. R. Chillarege and N. Bowen, "Understanding Large System Failures – A Fault Injection Experiment", Proc. FTCS-19, June 1989, pp. 356-363.
12. J. Güthoff and V. Sieh, "Combining Software-Implemented and Simulation-Based Fault Injection Into a Single Fault Injection Method", Proc. FTCS-25, June 1995, pp. 196-206.
13. A. Benso, M. Rebaudengo, L. Impagliazzo, P. Marmo, "Fault-List Collapsing for Fault Injection Experiments", RAMS 98, January 1998, pp. 383-388.
14. T. Tsai, M.-C. Hsueh, H. Zhao, Z. Kalbarczyk, R. Iyer, "Stress-Based and Path-Based Fault Injection", IEEE Transactions on Computers, Vol. 48, No. 11, November 1999, pp. 1183-1201.
15. L. Berrojo, I. González, F. Corno, M. Reorda, G. Squillero, L. Entrena, C. Lopez, "New Techniques for Speeding-up Fault-injection Campaigns", Proc. DATE 2002, March 2002, pp. 847-852.
16. J. Arlat, J.-C. Fabre, M. Rodríguez, F. Salles, "Dependability of COTS Microkernel-Based Systems", IEEE Transactions on Computers, Vol. 51, No. 2, February 2002, pp. 138-163.
17. Motorola Inc., "RISC Central Processing Unit Reference Manual", revision 1, 1999.
18. R. Barbosa, J. Vinter, P. Folkesson and J. Karlsson, "Fault Injection Optimization through Assembly-Level Pre-Injection Analysis", Tech. Report No. 04-07, Dept. of Comp. Eng., Chalmers University of Technology, Göteborg, Sweden, 2004.
19. winIDEA – iSystem's Integrated Development Environment, http://www.isystem.se/products/ide.htm,
January 24th, 2005.
20. PHYTEC Technology Holding Company, http://www.phytec.com/sbc/32bit/pc565.htm, January 24th, 2005.
21. iSYSTEM AB, http://www.isystem.se, January 24th, 2005.
22. iC3000 Active Emulator, http://www.isystem.se/products/emulators.htm#three, January 24th, 2005.

A Data Mining Approach to Identify Key Factors in Dependability Experiments

Gergely Pintér[1], Henrique Madeira[2], Marco Vieira[2], István Majzik[1], and András Pataricza[1]

[1] Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
[2] CISUC, University of Coimbra
{pinterg, pataric, majzik}@mit.bme.hu
henrique@dei.uc.pt, mvieira@isec.pt

Abstract. Our paper presents a novel approach for identifying the key infrastructural factors determining the behavior of systems in the presence of faults by the application of intelligent data processing methods on data sets obtained from dependability benchmarking experiments. Our approach does not rely on a-priori assumptions or human intuition about the dominant aspects enabling this way the investigation of highly complex COTS-based systems. The proposed approach is demonstrated using a commercial data mining tool from IBM on the data obtained from experiments conducted using the DBench-OLTP dependability benchmark. Results obtained with the proposed technique identified important key factors impacting performance and dependability that could not have been revealed by the dependability benchmark measures.

1 Introduction

It is widely recognized that the evaluation of dependability features in computer systems is a complex task. Traditional techniques based on analytical and simulation models have to be complemented with experimental approaches based on measurements taken from prototypes and (when possible) from real systems in the field. These experimental techniques, including fault injection, robustness testing, and field measurements, have been extensively used to evaluate specific fault tolerance mechanisms, validate robustness of software components, or to assess the general impact of faults in systems.

In spite of the big diversity of techniques and tools now available, all the experimental dependability evaluation approaches share a common problem: they tend to produce a large amount of raw data that have to be processed to obtain the desired dependability measures or to get useful information on how the target systems behave in the presence of faults. Very often the analysis of the experimental data is quite complex, as it has to take into account many aspects of the experimental setup such as the target system architecture and configuration, the workload, the type of faults involved, the environmental aspects, etc. Surprisingly, the problem of coping with the large size of the experimental data sets and the high complexity of the data analysis has received less attention in the dependability research effort. Researchers have focused on the development of fault injection and robustness testing tools and on the mitigation of problems such as experiment representativeness, intrusiveness and portability of tools, just to name some of them, and only a few papers deal with experimental data analysis, e.g. [Pataricza 2001, Pintér 2003, Madeira 2003].

In fact, many dependability evaluation tools such as fault injection and robustness testing tools have been proposed (e.g., Ferrari [Kanawati 92], Mafalda [Rodríguez 99], Xception [Carreira 98], Balista [Koopman 97], NFTAPE [Stott 00], GOOFI [Aidemark 01]) but all these tools either provide rudimentary means to analyze data or, more frequently, just store the raw results in a spreadsheet format. Although this approach can be acceptable for very specific (and simple) analysis, it is clearly not enough when the analysis required is complex or when the amount of raw data is very large.

A recent newcomer to the bag of experimental dependability evaluation techniques is the dependability benchmarking family. This new approach actually represents an attempt to standardize experimental techniques with the goal of comparing dependability features of different systems or components. This research effort has already caught the attention of companies such as Sun Microsystems [Zhu 2003a], IBM [Lightstone 2003] and Intel [Constantinescu 2003], and lead to many dependability benchmark proposals, covering domains such as transactional systems [Vieira 2003b, Buchacker 2003, Mauro 2004], web servers [Durães 2004b], and operating systems [Kalakech 2004]. Even topics such as human faults [Brown 2004] or hardware maintenance [Zhu 2003] have already been subject of dependability benchmark proposals.

Dependability benchmarks represent a new and important source of raw experimental data but the problem of analyzing that data has been even more neglected than in traditional fault injection and robustness testing. In fact, dependability benchmarks rely on a typically small set of measures and the data collected during the benchmark runs is just used to calculate the measures defined in the benchmark specification.

Furthermore, dependability benchmarking (and traditional fault injection as well) relies on *a-priori assumptions* about what are the measures we would like to improve (response time, throughput, availability etc.) and the benchmark performer should know what are the *infrastructural attributes* that determine these measures (e.g., CPU performance, disk bandwidth, operating system), as her/his goal is to tune the system under benchmark to deliver the best performance and dependability. Although this approach has been beneficially applied for improving systems of relatively low complexity it does not scale well to complex systems actually used in real applications.

In order to overcome this issue an *automated mechanism* is needed that supports the identification of key infrastructural factors by highlighting the potentially interesting phenomena in the large experiment database. On one hand this approach eliminates the need for a-priori human knowledge; on the other hand it avoids some bias coming from some human belief.

Our paper proposes a novel approach for identifying the key infrastructural factors determining the behavior of systems in the presence of faults by the application of *intelligent data processing methods* that have already been successfully applied in the business field for extracting previously unknown knowledge from large databases. The key idea of our approach is to perform benchmarking experiments on multiple configurations by applying different implementations of the same COTS component (e.g., different hardware setups, operating systems) and record as much information as possible about the infrastructure and the delivered performance and dependability attributes. On the basis of this information *data mining experiments* are carried out to identify which infrastructural factors were really relevant enabling the developers to improve the system without a-priori assumptions.

The structure of the paper is as follows: after providing an *overview about data mining* (Sect. 2) we briefly describe the *experiment setup* and the key *benchmark components* acting as the source of experimental data investigated in our work (Sect. 3). Sect. 4 discusses how to apply data mining for identifying the key factors that determine the behavior of systems in the presence of faults. Our observations in case of the DBench-OLTP experiment are discussed in Sect. 5. Finally Sect. 6 concludes the paper and outlines the directions of future research.

2 Background on Data Mining

Data mining is usually defined as an interdisciplinary field bringing together techniques from machine learning, pattern recognition, statistics, databases, and visualization to address the issue of *extracting* previously unknown, valid and actionable *information from large databases* to be used for making crucial business decisions [IBM 1999]. Our approach aims at porting data mining from the business field to the dependable computing domain for exploiting its benefits for *automatic identification of key factors that determine specific performance and dependability attributes of systems in presence of faults.*

Methods of data mining can be grouped in three families:

- *Predictive modeling* resembles the human learning experience, where we learn how to classify real-world objects into abstract categories by identifying the essential underlying characteristics of phenomena amongst the possibly high number of less important attributes. For example young children learn how to classify animals as cats and dogs by realizing that however animals are characterized by very large number of attributes (size, color, body structure, etc.) and many of them are not specific to any classes (e.g., there are cats and dogs of the same color, size) there are some key factors that can be used for assigning them to classes (e.g., the voice, body structure). The goal of predictive modeling is to build similar models by analyzing a teaching data set and identifying the attributes and their relations that represent the key factors for classifying database records using statistical methods. A typical business situation for building predictive models is when a company is interested in understanding the key aspects of customer behavior (e.g., which customers are going to leave the company) by identifying the dominant attributes (age, purchased products etc.) [IBM 1999].
- *Database segmentation* aims at partitioning a database in segments of similar records i.e., ones that share a number of properties and so are considered to be homogeneous. A typical business application of database segmentation is the identification of typical customer groups (e.g., highly paid urban women, male university students) to be addressed appropriately.

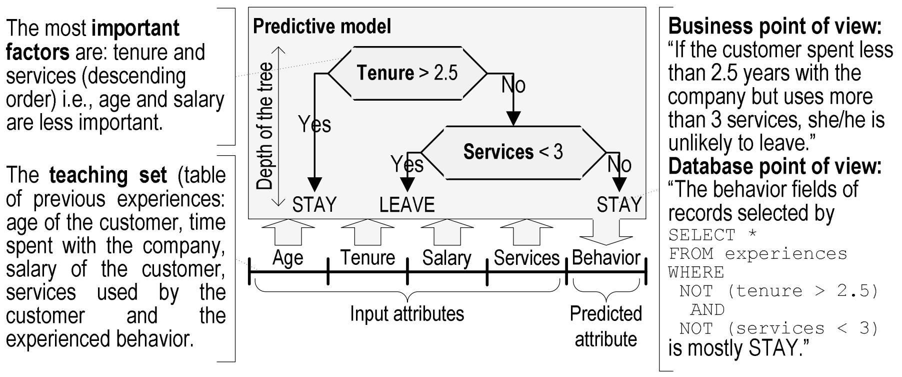
- The goal of *link analysis* is to establish links (associations) between individual records or sets of records in the database. In business applications link analysis is typically used for identifying products that tend to be sold together (market basket analysis) or for understanding long-term customer behavior for planning timely promotions etc.

Since our goal is to automatically identify key factors that determine specific attributes of systems in presence of faults, we selected the *classification method,* which is an implementation of the *predictive modeling* technique.

Classification aims at establishing a *specific class* for each record in a database. The class must be one from a finite set of possible and predetermined class values. The input of the method is a teaching data set that presents the correct answer for some already solved cases; the output is a *decision tree* where leaves are the *predicted classes* and the internal nodes are *atomic decisions (i.e., very simple predicates involving a single attribute e.g., "the color of the persons hair is black", "the age of the person is below 14").* The *key attributes* identified by the algorithm are this way in the predicates of the atomic decision nodes.

A typical business application example [IBM 1999] is depicted in Fig. 1. an insurance company interested in understanding the increasing rates of customer attrition. A predictive model has determined that the two attributes of interest are: the length of time the client has been with the company (Tenure) and the number of services that the client uses (Services). The decision tree presents the analysis in an intuitive way. Having built the decision tree on the basis of *previously experienced* behavior, the company can use it to *predict* the future behavior of its current customers and try to convince the ones who are likely to leave to stay with the company with special advertisement campaigns etc.

It is important to highlight that although the usual business application of the classification method is to *predict future behavior*, our goal is somewhat different: we don't want to *predict* anything (i.e., to *use* the algorithm built by the method) but we are interested in the *classification algorithm itself.* We will use the tree built by the method to *recognize dominant factors* (i.e., the attributes in the decision nodes).



The most **important factors** are: tenure and services (descending order) i.e., age and salary are less important.

The **teaching set** (table of previous experiences: age of the customer, time spent with the company, salary of the customer, services used by the customer and the experienced behavior.

Business point of view: "If the customer spent less than 2.5 years with the company but uses more than 3 services, she/he is unlikely to leave."

Database point of view: "The behavior fields of records selected by

```
SELECT *
FROM experiences
WHERE
 NOT (tenure > 2.5)
  AND
 NOT (services < 3)
```

is mostly STAY."

Fig. 1. Decision tree built by the classification method

Technically speaking a decision tree corresponds to a graph of SQL SELECT commands on the teaching set. Each SELECT divides the actual data set in two subsets. Classes are assigned to the resulting subsets by identifying the *most frequently occurring value* of the predicted attribute (e.g., performing the SELECT corresponding to the "Services < 3" predicate on the data set selected by the "Tenure <= 2.5" predicate results in two subsets i.e., the one where the customers use less than three services and the one where use three or more; since most of the customers in the first subset left the company, the data miner assigned the subset to the LEAVE class etc. (Fig. 1). The *quality of the classification* can be measured by the *homogeneity* of the subsets i.e., how many records are in the subset that do not belong to the majority. The data miner tool uses statistical methods to find the SELECTs that result in as homogeneous subsets as possible with respect to the predicted attribute, identifying this way the *attributes* that *play key role* in determining the value of the predicted attribute.

The maximal depth of the tree can be restricted by the data analyst. Specifying *high maximal depth* enables the data miner to define *more complex queries* resulting in *more homogeneous subsets* (but representing smaller subsets of the teaching data set). This feature can be used as a *zooming facility*: when restricting the maximal depth to a low value only the most important factors are visible enabling the data analyst to draw *general conclusions,* while building a sophisticated tree allows an *in-depth investigation* into the relations of attributes.

Since the data miner does not understand the semantics of the attributes care should be taken on the *appropriate selection of the teaching set*:

- When two attributes (columns) are *semantically related* that may represent redundant information that may become misleading for the analysis (for simplicity reasons we call this *misleading information* in the paper). This typically arises when two columns represent some kind of *refinement relation* e.g., the car manufacturer and the model (e.g., "VW" and "Golf", "VW" and "Passat", "Renault" and "Clio"). In this situation the model typically determines the manufacturer, since manufacturers do not copy model names (i.e., the model "Golf" determines that the manufacturer of the car is "VW"). From the database point of view selecting all VW products can be carried out by simply selecting the "VW" value of the *manufacturer* column or by selecting all VW models (Golf, Passat etc.) using the *model* column. Note that the selected subsets are *totally the same* in both cases, but the second query is more difficult to understand: if you do not recognize that all the models are from VW you will not realize that essentially we were focusing on the *manufacturer*. Since the data miner does not understand the semantics of columns it is subject to defining this kind of hard to understand queries i.e., defining hard to understand atomic decisions.
- When the teaching set is *statistically inappropriate* the decisions drawn by the data miner may be wrong (with the human learning analogy when a child sees only black cats and white dogs she/he may get to the wrong conclusion that the most important attribute for classifying animals as dogs or cats is the color). This *missing information* situation can arise when processing data obtained from a non-exhaustive experiment campaign.

3 Experiment Setup

We used a data set obtained during the DBench[1] project for the data-mining experiments presented here. The experimental data set used in our study comes from experiments conducted with a specific dependability benchmark (the DBench-OLTP) developed to On-Line Transaction Processing (OLTP) systems. DBench-OLTP [Vieira 2003a, Vieira 2003b] uses the workload and the general approach of the industry standard TPC-C performance benchmark [TPC-C] and adds two new components: (1) a faultload to emulate faults and upsets experienced by OLTP systems in the field (in the experiments used in our study the faultload includes operator faults) and (2) a set of new measures meant to characterize the behavior of the system in the presence of the artificially introduced faults. These new measures include service (TPC-C transactions per minute) in the presence of faults, TPC-C cost related measures in the presence of faults and availability measures.

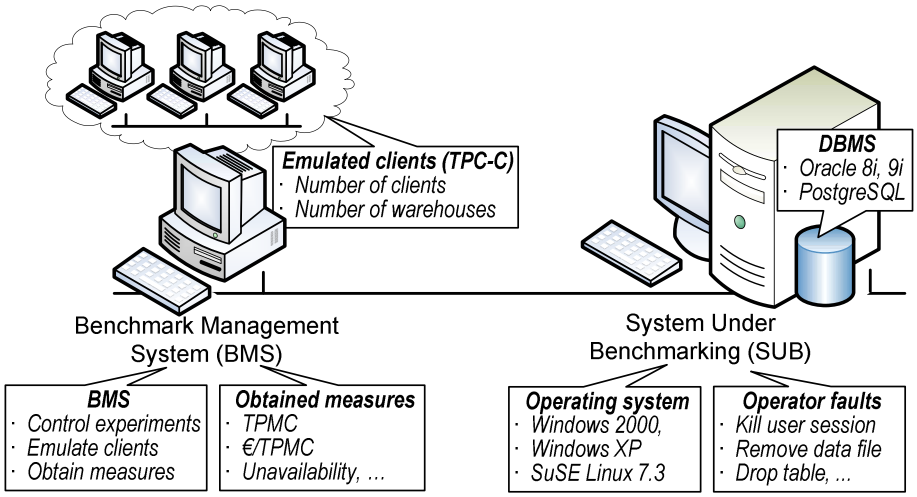


Fig. 2. DBench-OLTP experiment setup

The most important components of the DBench-OLTP benchmark are the *experiment setup*, the *workload*, the *faultload*, the *measures obtained* and the *benchmark procedure*:

- The main elements of the *experiment setup* (Fig. 2) are the *System Under Benchmark* (SUB) and the Benchmark Management System (BMS)[2] that emulates the client applications and records the data needed for calculating the benchmark measures. Several hardware-software configurations have been benchmarked with the DBench-OLTP: multiple hardware platforms, various operating systems (Win-

[1] See: http://www.criticalsoftware.com/DBench and http://www.laas.fr/DBench

[2] In the TPC-C terminology, the SUB is the System Under Test and the BMS includes the driver system (and some more features).

dows 2000, Windows XP, SuSE Linux 7.3) and database management systems (DBMS) (Oracle 8i, 9i, and PostgreSQL).

- The *workload* represents the work that the system must perform during the benchmark run. In the DBench-OLTP approach the workload of the TPC-C performance benchmark was used. TPC-C represents a business where a wholesale supplier has a number of warehouses and their associated sale districts and where users submit transactions that include entering and delivering orders, recording payments, checking the status of orders etc.
- The *faultload* represents the set of faults and stressful conditions that emulate real faults experienced by OLTP systems in the field. The DBench-OLTP approach focused on *operator faults* (i.e., mistakes of the DBMS administrator). The fault *types* injected during the experiments were as follows: abrupt operating system shutdown, abrupt DBMS shutdown, killing a set of user sessions, dropping a database table used by the workload, deleting the entire user schema, removing single data files from disk, removing sets of files from disk and removing all files from the disk. Since in several cases operator faults can only be detected by the system administrator (i.e., another operator) the *fault detection time* is not related to a system feature but an attribute of the faultload in this case. In DBench-OLTP experiments typical fault detection times were *estimated* taking into account the nature of the fault and field experience in OLTP system administration.
- The *measures* obtained characterize the *performance* and *dependability* of the SUB in presence of the faultload while executing the workload. The *performance related measures* include (1) the *TPMC* value (transactions executed per minute) calculated according to TPC-C (i.e., the total number of transactions per slot divided by the elapsed time) and (2) the *price per transaction (€/TPMC*, a ratio between the price and the performance of the SUB calculated according to TPC-C pricing rules). The *dependability related measures* include (1) the TPMC and €/TPMC in the presence of the faultload (measure the impact of the faults on the service provided by the OLTP system, (2) the *number of data errors* detected by consistency checking mechanisms measuring the impact of faults on the data integrity, (3) the *unavailability from the SUB point of view* (the SUB is considered to be available from its own point of view if it is able to respond to at least one client within the maximal response time), and (4) the *aggregated unavailability from the clients' point of view* (the SUB is considered to be unavailable from the clients' point of view if it is unable to respond within the maximal response time or returns an error). The data mining analysis presented in this paper focuses on two measures: the *number of transactions executed during a slot* (this raw performance attribute is the input for calculating the TPMC) and the *aggregated unavailability from the clients' point of view*.
- The *benchmark procedure* (Fig. 3) consists of *fault injection slots*. At the beginning of a slot the state of the SUB is *explicitly restored*. Measurements are performed with the system in a *steady state* condition i.e., after a given time (*steady state time*) that is enough for the system to achieve its maximum throughput (e.g., filling data caches). Having achieved the steady state the *fault is injected* after a certain amount of time (*injection time*). As discussed above the *fault detection time* is artificially defined (taking into account the nature of the fault and field experience in OLTP system administration). After detecting the fault *diagnostic* and

recovery procedures are initiated (the time needed for this is the *recovery time*). Having completed the recovery the workload is continued for a while (*keep time*) to measure the system's speedup after recovery. At the end of the slot *data integrity test* are performed.

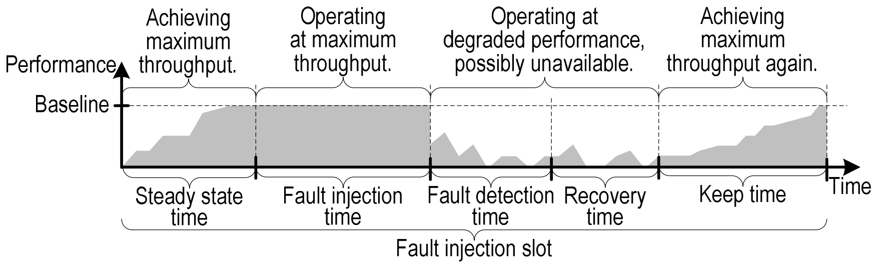

Achieving maximum throughput.
Operating at maximum throughput.
Operating at degraded performance, possibly unavailable.
Achieving maximum throughput again.
Performance
Baseline
Steady state time
Fault injection time
Fault detection time
Recovery time
Keep time
Time
Fault injection slot

Fig. 3. Fault injection slot

4 Applying Data Mining to Identify Dominant Factors

This section discusses how to exploit the benefits of data mining for identifying dominant factors that determine the behavior of the system under benchmarking in presence of faults. The idea of our approach is to process important performance and dependability measures obtained during the fault injection experiments by building classification trees and identifying the key factors by investigating the tree built by the data miner. The steps of our approach are as follows:

1. *Preparing the input data set* for making it appropriate to be analyzed by the data miner.
2. *Running the data miner* for building decision trees (i.e., applying the *classification method*).
3. Analyzing the trees for collecting the dominant factors and explaining the phenomena.

This section walks through these steps for the case of the DBench-OLTP data set processed in our experiments; finally the last subsection presents a short evaluation of our approach.

4.1 Preparing Input Data

The goal of the *data preparation* is to transform the legacy data obtained during the experiments of the DBench project to a format that is appropriate for using as a teaching set in data mining. This process involves (1) the re-organization of individual tables where the experimental data is stored for eliminating column correlations that may represent *misleading information*, and *joining tables* into a single view (required by the data miner tool), (2) eliminating the effects of missing information and (3) *defining classes* for the classification. Although the discussion below describes these steps as a human process for simplicity reasons, most of the steps may be facilitated by an automatic process.

Re-organization of Individual Tables and Defining a Flat View
The goal of re-organizing the individual tables is to eliminate the *semantic relation* of attributes (columns) that may represent *misleading information*. In case of the DBench-OLTP data we identified some attribute pairs that represent some kind of *refinement* relation, e.g., the *version* of the operating system (e.g., "2000", "XP", "7.3") can be considered as a *refinement* of the OS *family* ("Windows", "SuSE"): e.g., since SuSE has never released "SuSE 2000" the *version* attribute *determines* the *family*. The misleading information was eliminated by introducing a new *operating system* attribute instead of the individual family and version attributes containing the valid *combinations* (i.e., "Windows 2000", "Windows XP" and "SuSE 7.3").

Technically speaking, when re-organizing the individual tables we have to *answer* the following *question*: "Are there two columns A and B that we can define a SELECT operation above A and one above B resulting in the same subsets?" If there are ones, these columns may represent *misleading information* and should be *reorganized* e.g., by *merging* them as presented above. Note that this step requires expertise neither in database management nor in data mining: the question can be answered by the *designer of the experiments* (i.e., the benchmark expert) on the basis of a-priori knowledge about the *semantics* of the columns.

Since the data miner expects the teaching set as a *single* database *table* or *view* the tables of the legacy table structure of the DBench data was *joined* into a single view.

Dealing with Missing Information
Since we are using a legacy data set that was not collected for data mining purposes we have to take care of the *statistical relevance* of the teaching set, i.e., we should avoid drawing wrong conclusions resulting from *missing information*. For example the original goal of DBench-OLTP was to *compare* DBMSs in a *benchmarking* style (e.g., *compare* the two Oracle versions 8i and 9i on a Windows platform, *compare* Oracle and PostgreSQL on the SuSE platform) this way experiments were not carried out on some operating system – DBMS-OLTP combinations (e.g., PostgreSQL was not benchmarked on Windows). This way, considering the entire data set *as a whole* it should not be used as a teaching set since the *missing information* may obscure the observations.

Obviously the best solution would be to *provide information* by carrying out the missing experiments, but unfortunately this approach is not viable in several cases e.g., since the PostgreSQL is available for the Linux platform *only*. In order to *eliminate the effect* of the missing information we focused on a subset of data that was exhaustive with respect to some restriction (e.g., by focusing on different DBMSs on the SuSE platform).

Technically speaking, when searching for missing information we have to *answer* the following *question*: "Are there experiments that were not carried out on specific platforms?" If there are ones, the missing experiments should be performed or data subsets have to be selected that are exhaustive with respect to some restrictions, e.g., focusing only on those platforms where all experiments were carried out. Note that this step requires expertise neither in database management nor in data mining: the question can be answered by the expert who *performed the experiments* on the basis of a-priori knowledge about the *experiments* loaded into the database.

Defining Classes for Numeric Measures
Since the measures obtained in the experiments were *numeric variables* (e.g., server unavailability experienced by the clients in seconds) and the classification method requires that the predicted attribute is from a finite set of possible *predetermined class values* we had to *define classes* and *assign numeric values* to the corresponding *classes* in the teaching set. This was carried out by dividing the intervals of the obtained measure in sub-intervals (i.e., corresponding to classes like "very low unavailability", "low unavailability"). For example, in case of a data subset selected in the previous step the attribute "total number of transactions executed per slot in presence of faults" was between 0 and 70000, this way we introduced seven sub-intervals of the width of 10000 (e.g., [0-9999], [10000-19999]...) for classification purposes. Since the chosen *class granularity* (i.e., the number of sub-intervals) was mainly intuitive we carried out a *validation experiment* to identify the *impact* of *class granularity* to the trees (see the discussion at the end of this section).

4.2 Mining Data

Having prepared the input data configuring the data miner[3] is a few steps:

1. Selecting the data mining method (classification from the predictive modeling family).
2. Defining the data source to be processed (i.e., selecting the view containing the prepared data).
3. Configuring the classification: (1) selecting the *input columns* (i.e., the ones that possibly contain interesting factors: all columns should be selected that describe some aspects of the measurement setup, hardware-software infrastructure, workload, faultload etc.) and (2) the *attribute to be predicted* (i.e., the performance or dependability measure to be analyzed e.g., "server unavailability experienced by the clients in seconds", "TPMC in presence of faults"). Analyzing our aggregated data sets containing some thousand records by the classification method took a few seconds on an ordinary desktop PC. The result of the operation is presented as an *interactive decision tree*: sub-trees can be closed and opened, leaf data distribution can be displayed etc. Although interactivity supports the analysis, screenshots are not suitable for being inserted into a paper (because they miss some details and occupy a lot of space), thus the figures presented here were redrawn and annotated in a drawing tool.

5 Result Discussion and Analysis

This subsection presents how to *identify* interesting phenomena by the classification method and how to *explain* the phenomena by field expertise. The discussion is organized as follows:

- Data mining aims at automatically *identifying phenomena*. In this discussion we will (1) first *textually* formulize a *question*, (2) *configure* the data miner accord-

[3] We used IBM DB2 Intelligent Miner for Data 6.1 for the data mining experiments and IBM DB2 Universal Database Version 8 for storing the experiment data.

ingly (3) *explain* the *notation* of the figures (decision trees) and (4) *textually formulize* the *atomic decisions* chosen by the miner and *identify* the *key factors* determining the value of the specific predicted attribute.

- Having automatically identified the phenomena the *explanation* requires field expertise i.e., in our case knowledge about the internal operation of databases, fault injection etc. In this discussion we present the explanation of some phenomena identified by the data miner that probably would not have been identified when relying barely on human intuition.

5.1 Identifying Phenomena by Classification Tree Analysis

In cases of the two experiments presented here we were using a *data subset* that was exhaustive with respect to restricting the operating system to Microsoft Windows 2000. Since the PostgreSQL was not available for the Windows platform we had data about *two Oracle versions* (8i and 9i) set up according to two inherently different *configurations* this way we could investigate the effects of the *DBMS version* (8i and 9i), the *DBMS configuration* and the *fault-load* on the performance and dependability measures.

In the *first experiment* presented here (Fig. 4) we aimed at identifying the key factors determining the *total number of transactions per slot* successfully performed during a fault injection slot. Informally, we asked the data miner to *answer the following question*: "What are the dominant factors that determine the performance of Oracle databases running on Windows 2000 as expressed by the total number of successful transactions per slot?" Obviously questions like that naturally emerge when planning databases that should deliver high performance even in presence of faults. Answering this question will suggest which DBMS version is to be used, how to configure it and which faults are the most serious ones in performance aspects.

The data miner was *configured* according to the discussion above: the *input columns* were the version and configuration of the DBMS and the attributes of the fault-load (type, injection and detection time, etc.). The *predicted attribute* was the class of the "total number of successful transactions per slot" measure. Since the total number of transactions per slot fall in the [0-70000] interval in this subset of the teaching set we *defined* seven *classes* ([0-9999], [10000-19999]… etc.).

The *notation* of Fig. 4 is quite straightforward: the predicted classes are visualized by *highlighting the appropriate sub-interval* in the small graphs in the leaves (where the class homogeneity was considered to be low multiple sub-intervals were highlighted). The numbers under the small graphs in the leaves indicate the size of the corresponding data subsets (e.g., "When using configuration Conf-A and the fault injection time is above 870 (stepping always to the No edge from the root) the total number of transactions per slot was between 40000 and 70000 (very high) and this selection corresponds to 54 records in the teaching set.").

The *atomic decisions* are *textually formalized* in the callout texts in Fig. 4. The *key factors* that determine the total number of successful transactions per slot in presence of faults can be easily *identified* since the most important one is in the root of the tree; the less important ones are ever closer to the leaves. This way the data miner determined that the *most important factor* determining the total number of successful transactions per slot in presence of faults in case of the Oracle databases running on the Windows 2000 platform was the *configuration* (Conf-A was found to deliver higher performance); another attribute of high impact is the *time of fault injection* (the later we inject the fault, the higher the total number of transactions per slot), and the actual *fault load* has somewhat smaller impact (dropping a table is a more serious fault in this aspect than removing files from the disk or abruptly shutdown). Note that these observations were drawn only by investigating the tree without a-priori knowledge about the system under benchmarking—we have only *identified* some phenomena until now, but we have not yet *explained* them. The explanation of the phenomena requires the knowledge of an OLTP expert.

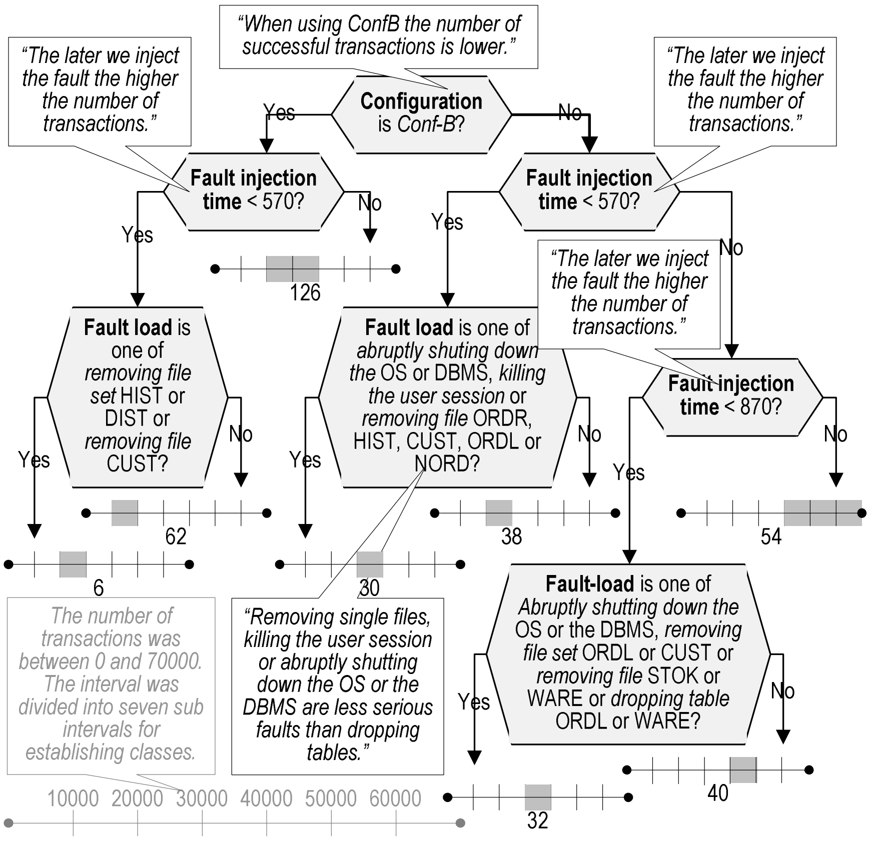

"When using ConfB the number of successful transactions is lower."
Configuration is Conf-B?
Yes
No
"The later we inject the fault the higher the number of transactions."
Fault injection time < 570?
Fault injection time < 570?
"The later we inject the fault the higher the number of transactions."
126
"The later we inject the fault the higher the number of transactions."
Fault load is one of removing file set HIST or DIST or removing file CUST?
Fault load is one of abruptly shuting down the OS or DBMS, killing the user session or removing file ORDR, HIST, CUST, ORDL or NORD?
Fault injection time < 870?
62
38
54
6
30
"Removing single files, killing the user session or abruptly shutting down the OS or the DBMS are less serious faults than dropping tables."
The number of transactions was between 0 and 70000. The interval was divided into seven sub intervals for establishing classes.
Fault-load is one of Abruptly shutting down the OS or the DBMS, removing file set ORDL or CUST or removing file STOK or WARE or dropping table ORDL or WARE?
40
32
10000 20000 30000 40000 50000 60000

Fig. 4. Total number of transactions per slot in the presence of faults performed by Oracle databases running on Windows 2000

In the *second experiment* presented here (Fig. 5) we aimed at identifying the key factors determining the *server unavailability experienced by the clients* (i.e., the sum of seconds while the individual clients found the server to be *unavailable*). Informally, we asked the data miner to *answer the following question*: "What are the dominant factors that determine the unavailability of Oracle databases running on

Windows 2000 from the clients' point of view?" Obviously questions like that naturally emerge when planning databases that should be highly available even in presence of faults. Answering this question will suggest which DBMS version is to be used, how to configure it and which faults are the most serious ones in availability aspects.

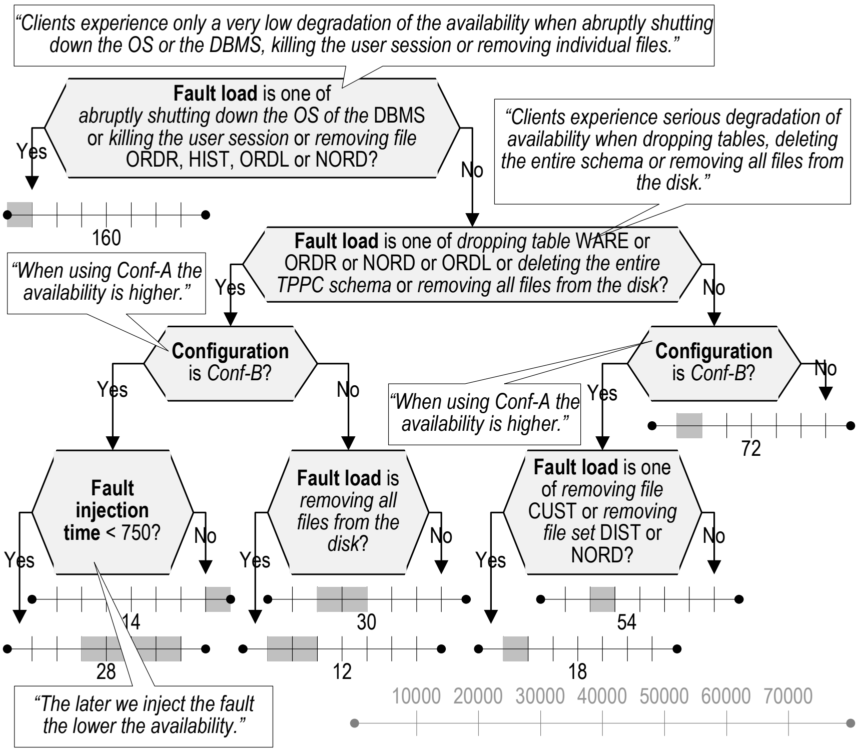

"Clients experience only a very low degradation of the availability when abruptly shutting down the OS or the DBMS, killing the user session or removing individual files."
Fault load is one of abruptly shutting down the OS of the DBMS or killing the user session or removing file ORDR, HIST, ORDL or NORD?
Yes
No
160
"Clients experience serious degradation of availability when dropping tables, deleting the entire schema or removing all files from the disk."
Fault load is one of dropping table WARE or ORDR or NORD or ORDL or deleting the entire TPPC schema or removing all files from the disk?
"When using Conf-A the availability is higher."
Configuration is Conf-B?
"When using Conf-A the availability is higher."
Configuration is Conf-B?
72
Fault injection time < 750?
Fault load is removing all files from the disk?
Fault load is one of removing file CUST or removing file set DIST or NORD?
14
30
54
28
12
18
"The later we inject the fault the lower the availability."
10000 20000 30000 40000 50000 60000 70000

Fig. 5. Unavailability of Oracle databases running on Windows 2000 experienced by clients

The data miner was *configured* similarly to the previous experiment but the *predicted attribute* was obviously the class of the "number of seconds clients experienced server unavailability" measure. Since the number of seconds fall in the [0-80000] interval in this subset of the teaching set we *defined* eight *classes* ([0-9999], [10000-19999]… etc.).

The *atomic decisions* are *textually formalized* in the callout texts in Fig. 5. The data miner determined that the *most important factor* that determines the unavailability of Oracle databases running on Windows 2000 in presence of faults is the *type of the faults injected*. The tree highlights that there are *less serious faults* like abruptly shutting down the DBMS or the operating system or *killing the user session,* while inadvertently *dropping tables* results in *high unavailability time*. *Configuration* of the DBMS is also an important attribute: we can achieve *higher availability* by using

Conf-A under the same circumstances. Note that although there were experiments where the newer Oracle DBMS (version 9i) was proven to deliver higher performance and expose better dependability characteristics, in case of the two experiments presented here *there were no important differences between the two versions*. This observation can also be important when considering software upgrade for improving system performance or availability.

5.2 Explaining the Phenomena by Field Expertise

Although data mining provides a powerful support by *automatically identifying* key factors determining the values of performance and dependability attributes, the phenomena should be *explained* by a *field expert* for turning the observations into financial, performance or dependability benefits. The following points provide explanation examples (for space reasons only some of the identified phenomena were selected):

- The *impact of the configuration* can be explained by the fact that Conf-A is an effective well-tuned configuration while Conf-B restricts the size of the redo log files to 1MB (very low size preventing the DBMS from exploiting the performance benefits of postponed write operations and buffering) and sets the checkpoint interval to 4 seconds (very low time forcing the DBMS to synchronize the memory image and the data files uselessly often). The *performance benefits* (higher number of successful transactions per slot in case of Conf-A) are this way obvious. Concerning availability, although Conf-B should favor fast recovery (because the high checkpointing rate) this is true for some types of faults only. Globally, the best performance of the well tuned Conf-A also favors availability.
- Both experiments highlighted the important *impact of the faultload*: dropping tables proved to be more serious than other faults (removing files, abrupt shutdown etc.) in both *performance* and *availability* aspects (i.e., results in serious degradation of the total number of successful transactions per slot and availability). These phenomena can be explained by the *recovery mechanism* of the DBMS: removed data files can be restored by copying back the latest backup file and re-executing the transactions involving the data file that were performed until the creation of the backup. Since during the reconstruction of files the DBMS *can accept transactions* that are not related to the file deleted (note that, in some cases a single table was partitioned by four files) the clients may not experience unavailability at all. Restoring inadvertently dropped tables is more complicated since dropping a table is a valid database operation. This way the only solution is to restore the latest backups of *all the data files* and re-execute *all the transactions* that were performed between creating the backup and dropping the table. This way restoring dropped tables is not only more time consuming but prevents the DBMS from accepting transactions during the recovery process resulting in serious degradation of the availability.
- The *impact of the fault injection time* is probably one of the most interesting observations. The two phenomena identified by the miner may seem to be contradictory for the first sight: the later we inject the fault the *higher the total number of successful transactions per slot* but the *lower the availability*. In order to explain these phenomena we have to take into consideration the performance graph of the DBMS (Fig. 6). Before injecting the fault the server is operating at the baseline

performance; obviously *the later we inject the faults the greater the total number of transactions per slot performed during this period.* After the fault injection the server may be unavailable during the fault detection and recovery time (depending on the fault type). Although the function is probably not linear it is quite obvious that *the greater the total number of transactions per slot executed before the fault injection the longer the time needed for recovery* this way we transitively explained the observation that *the later we inject the fault the lower the availability.* Having finished the recovery the DBMS is fully functional again but the situation is similar to the one during the steady state time: the DBMS needs some time to achieve its maximum throughput again, this way the delivered performance is *below the baseline performance* for a while. If this time is high, the total number of successful transactions per slot is dominated by the ones performed *before* the fault injection (the total number of transactions per slot is proportional to the size of the area under the performance graphs marked gray in Fig. 6). This explains the observation that *the later we inject the fault the higher the number of successful transactions.*

The phenomena *identified* by the miner and *explained* by field experts can be used for *improving the performance and the dependability* of the systems: e.g., by configuring the DBMS effectively, introducing fault tolerance measures against the most serious faults.

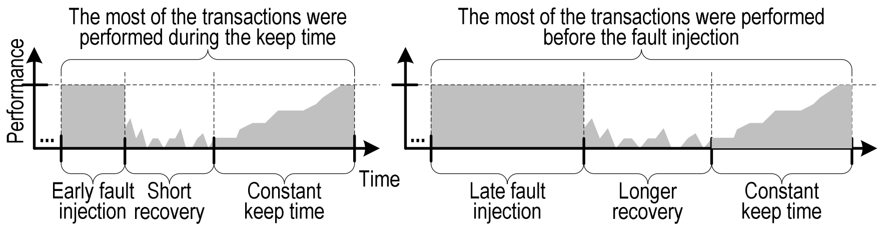

The most of the transactions were performed during the keep time
The most of the transactions were performed before the fault injection
Performance
Time
Early fault injection
Short recovery
Constant keep time
Late fault injection
Longer recovery
Constant keep time

Fig. 6. Comparison of early and late fault injections

5.3 Evaluating the Impact of Class Granularity

Since the definition of classes (sub-intervals) enabling the classification tree analysis was mainly intuitive (i.e., how many sub-intervals to be introduced) we were interested in *evaluating* the *impact of the class granularity* (number of classes) on the resulting decision trees. We set up different experiments focusing on the same measure but defining classes of different granularity. For example the "TPMC in presence of faults" performability attribute fall in the [0-2000] interval in the case of selecting all database managers (two Oracle versions and PostgreSQL) running on the SuSE platform; this way we set up experiments with classes corresponding to sub intervals of width of 1000, 500, and 250 respectively (the trees corresponding to the first two granularity levels are presented in Fig. 7). According to our observations the role of class granularity is similar to restricting the dept of the classification tree i.e., can be seen as a zooming facility:

- *Low interval granularity* (wide sub-intervals, low number of classes) results in *small trees* that enable *general observations*: in case of the “TPMC in presence of faults” attribute the tree built after dividing the entire interval into two sub-intervals highlights the DBMS in use to be the most important factor determining the performance of the system and indicates Oracle versions more performable (Fig. 7 left side).
- Choosing ever *higher granularity* (narrower sub-intervals, higher number of classes) the *tree* will get ever *larger* enabling *in-depth investigation* of the phenomena (Fig. 7 right side): in case of the “TPMC in presence of faults” attribute the tree built after dividing the entire interval into four sub-intervals still highlights the DBMS in use to be the most important factor determining the performance but also indicates that Oracles are not only *faster* but are better in *fault-tolerance aspects*: while in case of PostgreSQL only the “killing the user session” fault can be considered as less serious (i.e., resulting in low degradation of the performance) in case of Oracles all the “killing the user session”, “abruptly shutting down the DBMS” and “removing single files” can be considered as less serious.

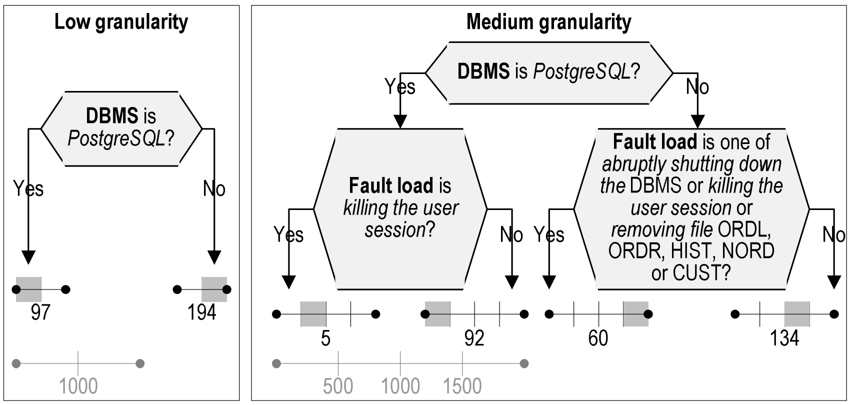

Low granularity
DBMS is PostgreSQL?
Yes
No
97
194
1000
Medium granularity
DBMS is PostgreSQL?
Yes
No
Fault load is killing the user session?
Yes
No
Fault load is one of abruptly shutting down the DBMS or killing the user session or removing file ORDL, ORDR, HIST, NORD or CUST?
Yes
No
5
92
60
134
500
1000
1500

Fig. 7. Impact of class granularity on the decision trees in case of TPMC delivered by Oracle and PostgreSQL databases on SuSE 7.3 in presence of faults

6 Conclusions and Future Work

Our paper has presented a novel approach for automatically identifying the key factors determining the behavior of systems in the presence of faults by intelligent data processing methods. We have discussed how to exploit the benefits of automated tools enabling this way the application of most advanced data processing intelligence for analyzing experiment results by the dependability community. We have proven the viability of our approach by analyzing a data set obtained during a standard dependability benchmarking experiment. In the near future we would like to investigate the benefits of the *combined application* of data mining and on-line analytical processing (OLAP, [Madeira 2003]). Furthermore we would like to apply data mining for *validating benchmarks* by proposing a method for ensuring that the measures delivered by the benchmark are really determined by the infrastructural attributes they are meant to characterize.

Acknowledgment

Funding for this paper was provided, in part, by the Portuguese Government/European Union through the R&D Unit 326/94 (CISUC) and GRICES, by the Hungarian-Portuguese Intergovernmental S&T Cooperation Programme P-19/03, by OTKA T038027 from the Hungarian NSF, and by an IBM Faculty Award. The authors also acknowledge the software licenses used for this research offered by the IBM Scholars Program.

References

[Aidemark 01] J. Aidemark, J. Vitter, P. Folkesson, and J. Karlsson, "GOOFI : Generic Object-Oriented Fault Injection Tool", Int. Conference on Dependable Systems and Networks (DSN-2001), (Göteborg, Sweden), pp. B102-B103, Chalmers University of Technology, Göteborg, Sweden, July 1-4, 2001.

[Brown 2004] A. Brown, L. Chung, W. Kakes, C. Ling, D. A. Patterson, "Dependability Benchmarking of Human-Assisted Recovery Processes", Dependable Computing and Communications, DSN 2004, Florence, Italy, 2004

[Buchacker 2003] K. Buchacker, M. Dal Cin, H.-J. Hoexer, R. Karch, V. Sieh, and O. Tschaeche, "Reproducible Dependability Benchmarking Experiments Based on Unambiguous Benchmark Setup Descriptions", The Int. Conference on Dependable Systems and Networks, DSN-PDS2003, San Francisco, CA, June 22 - 25, 2003.

[Carreira 98] J. Carreira, H. Madeira, and J. G. Silva, "Xception: Software Fault Injection and Monitoring in Processor Functional Units", IEEE Transactions on Software Engineering, vol. 24, no. 2, February 1998.

[Constantinescu 2003] C. Constantinescu, "Experimental Evaluation of Error-Detection Mechanisms", IEEE Transactions on Reliability, Vol. 52, No. 1, March, 2003, pp. 53-57.

[Durães 2002] J. Durães and H. Madeira, "Characterization of Operating Systems Behaviour in the Presence of Faulty Drivers Through Software Fault Emulation", in Proc. 2002 Pacific Rim Int. Symp. on Dependable Computing (PRDC-2002), Tsukuba, Japan, 2002.

[Durães 2004] J. Durães, V. Marco. and H. Madeira, "Dependability Benchmarking of Web-Servers", The 23rd International Conference of Computer Safety, Reliability and Security, SAFECOMP 2004, Potsdam, Germany, 2004.

[IBM 1999] IBM, "Intelligent Miner for Data, Applications Guide", 1999.

[Kalakech 2004] A. Kalakech, T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun, "Benchmarking Operating System Dependability: Windows 2000 as A Case Study", The 10th Intl. Symp. Pacific Rim Depend. Computing, PRDC2004, Tahiti, French Polynesia, March 03-05, 2004.

[Kanawati 92] G. Kanawati, N. Kanawati, and J. Abraham, “FERRARI: A Tool for the Validation of System Dependability Properties”, Proc. of the 22th IEEE Fault Tolerant Computing Symp., FTCS-22, pp. 336-344, June 1992.

[Koopman97] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, T. Marz, “Comparing Operating Systems using Robustness Benchmarks”, in Proc. 16th Int. Symp. on Reliable Distributed Systems, SRDS-16, Durham, NC, USA, 1997.

[Lightstone 2003] S. Lightstone, J. Hellerstein, W. Tetzlaff, P. Janson, E. Lassettre, C. Norton, B. Rajaraman, and L. Spainhower. "Towards Benchmarking Autonomic Computing Maturity.", 1st IEEE Conf. on Industrial Automatics (INDIN-2003), Banff, Canada, August 2003.

[Madeira 2003] H. Madeira, J. Costa and M. Vieira, "The OLAP and Data Warehousing Approaches for Analysis and Sharing of Results from Dependability Evaluation Experiments", IEEE/IFIP International Conference on Dependable Systems and Networks, DSN2003, San Francisco, USA, June 22-25, 2003.

[Mauro 2004] J. Mauro, J. Zhu and I. Pramanick. “The System Recovery Benchmark,” in Proc. 2004 Pacific Rim International Symposium on Dependable Computing (PRDC 2004), Papeete, Polynesia, IEEE CS Press, 2004.

[Pataricza 2001] A. Pataricza and B. Tolvaj. “Data mining techniques in the experimental analysis of dependability”, in Proc. IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2001),, pp. 273-280, Győr, Hungary, 2001.

[Pintér 2003] G. Pintér and A. Pataricza. “Data Mining in Fault Injection”, in Proc. IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS 2003), pp. 307-308, Poznan, Poland, 2003.

[Rodríguez 99] M. Rodríguez, F. Salles, J.-C. Fabre and J. Arlat, “MAFALDA: Microkernel Assessment by Fault Injection and Design Aid”, in Proc. 3rd European Dependable Computing Conf. (EDCC-3), (J. Hlavicka, E. Maehle, A. Pataricza, Eds.), Prague, Czech Republic, LNCS, 1667, pp.143-60, Springer, 1999.

[Stott 00] D.T. Stott, B. Floering, Z. Kalbarczyk, R.K. Iyer, "Dependability Assessment in Distributed Systems with Lightweight Fault Injectors in NFTAPE," Proc. IEEE Int'l Computer Performance and Dependability Symp. (IPDS 2000), pp.91-100, March 2000.

[Vieira 2003a] M. Vieira and H. Madeira, "Benchmarking the Dependability of Different OLTP Systems", IEEE/IFIP Int. Conference on Dependable Systems and Networks, DSN2003, San Francisco, USA, June 22-25, 2003.

[Vieira 2003b] M. Vieira, and H. Madeira, “A Dependability Benchmark for OLTP Application Environments”, 29th International Conference on Very Large Databases, VLDB 2003, Berlin, Germany, Sept. 9-12, 2003.

[Zhu 2003a] J. Zhu, J. Mauro, and I. Pramanick, “Robustness Benchmarking for Hardware Maintenance Events”, in Proc. Int. Conf. on Dependable Systems and Networks (DSN 2003), pp. 115-122, San Francisco, CA, IEEE CS Press, 2003.

[Zhu 2003b] J. Zhu, J. Mauro and I. Pramanick. “R3 - A Framework for Availability Benchmarking,” in Proc. Int. Conf. on Dependable Systems and Networks (DSN 2003), pp. B-86-87, San Francisco, CA, USA, 2003.

PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis

Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger

CEA/Saclay, DRT/LIST/SOL/LSL,91191 Gif sur Yvette, France
{Nicky.Williams, Bruno.Marre, Patricia.Mouy, Muriel.Roger}@cea.fr

Abstract. We present the PathCrawler prototype tool for the automatic generation of test-cases satisfying the rigorous all-paths criterion, with a user-defined limit on the number of loop iterations in the covered paths. The prototype treats C code and we illustrate the test-case generation process on a representative example of a C function containing data-structures of variable dimensions, loops with variable numbers of iterations and many infeasible paths. PathCrawler is based on a novel combination of code instrumentation and constraint solving which makes it both efficient and open to extension. It suffers neither from the approximations and complexity of static analysis, nor from the number of executions demanded by the use of heuristic algorithms in function minimisation and the possibility that they fail to find a solution. We believe that it demonstrates the feasibility of rigorous and systematic testing of sequential programs coded in imperative languages.

1 Introduction

Rigorous testing of delivered software, by its implementers or by external certifiers, is increasingly demanded, along with some quantification of the degree of confidence in the software implied by the test results. The reasons for this include the increase in the deployment of embedded software systems and the re-use of off-the-shelf components. This sort of testing cannot be based on a restricted set of hand-crafted test objectives or use-cases, which may have to be manually updated if the software requirements change. Testing must be made as automatic as possible, with automatic generation of a large number of test-cases according to a well-justified selection criterion.

2 Related Work

There has been much research on the automatic generation of structural test-cases but most of it addresses the Test Data Generation Problem (TDGP) of finding data to cover a test objective in the form of given node, branch or path of the control flow graph.

```
void Merge (int t1[],int t2[],int t3[],int l1,int l2){
  int i = 0;   int j = 0;   int k = 0;
  while (i < l1 && j < l2) {
    if (t1[i] < t2[j]) {
      t3[k] = t1[i];
      i++; }
    else {
      t3[k] = t2[j];
      j++; }
    k++; }
  while (i < l1) {
   t3[k] = t1[i];
   i++;
   k++; }
  while (j < l2) {
   t3[k] = t2[j];
   j++;
   k++; }
}
```

Fig. 1. Source code of the function `Merge`

Static approaches to test-case generation [2, 3, 15] typically select a path from the control flow graph covering the test objective, derive the path predicate as a set of constraints on the input values and then solve these constraints to find a test-case which activates the path. In theory, symbolic execution can be used to construct the path predicate. However, in practice symbolic execution encounters problems in the detection of infeasible paths (notably in the case of loops with a variable number of iterations), the treatment of aliases and the complexity of the formulae which are gradually built up.

Dynamic approaches [1, 5, 9] avoid the problems of symbolic execution by dispensing with the path predicate and using general heuristic function minimisation techniques to modify the input data so that the test objective is covered. The first set of input data is arbitrarily selected and the program is instrumented so as to indicate the branches taken and evaluate their "distance" from the test objective. Function minimisation must reduce this distance to zero. The disadvantages of these techniques are that they may need a great many executions before a test-case is found, they may fail to find a test-case even when one exists and they do not terminate if the desired path is actually infeasible.

We address a different problem to that of most previous work, and adopt a different solution. We believe that rigorous testing is possible if sufficiently automated and we therefore base our work on a rigorous test criterion: 100% coverage of feasible execution paths. The TDGP is not the best formulation of this problem. We do not need to construct the control flow graph, enumerate all the paths in the graph, many of which will be infeasible, and search for a test for each. Instead, we iteratively cover "on the fly" the whole input space of the program under test. This is an extension of the idea sketched out in [14].

Like the dynamic approaches to test data generation, PathCrawler is based on dynamic analysis, but instead of heuristic function minimisation, it uses constraint logic programming to solve a (partial) path predicate and find the next test-case, as in the approaches based on static analysis. It suffers neither from the approximations and complexity of static analysis, nor from the number of executions demanded by heuristic algorithms used in function minimisation and the possibility that they fail to find a solution.

3 Our Approach

Our approach is applicable to all sequential programs coded in an imperative language and the prototype has been implemented for C. This paper extends [8], notably by illustrating the test generation process step-by-step on an example: the C function `Merge`, whose source code is shown in Fig. 1. `Merge` takes as input two arrays, `t1` and `t2`, of ordered integers and their effective lengths, `l1` and `l2`, and outputs, in array `t3`, all their elements, in order. `Merge` is representative of many of the problems posed by C code: it contains arrays of variable length, loops with a variable number of iterations and many infeasible paths and only produces the correct result if the input arrays are sorted. In this section we give an overview of our approach and in the following sections we describe its principal stages: Instrumentation, Substitution and Constraint Solving. We then describe the results of applying PathCrawler to the function `Merge`, before concluding with a discussion of further work.

Our approach (see Fig. 2) starts with the instrumentation of the source code so as to recover the symbolic execution path each time that the program under test is executed. The instrumented code is executed for the first time using a "test-case" which can be any set of inputs from the domain of legitimate values. The symbolic path which we recover is transformed into a path predicate which defines the "domain" of the path covered by the first test-case, i.e. the set of input values which cause the same path to be followed. The next test-case is found by

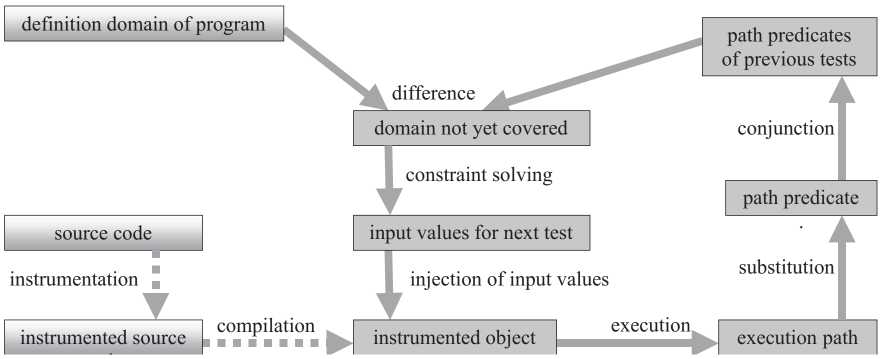


Fig. 2. Our approach

solving the constraints defining the legitimate input values outside the domain of the path which is already covered. The instrumented code is then executed on this test-case and so on, until all the feasible paths have been covered.

Loops with a variable number of iterations, such as the three loops in our example function `Merge`, can cause a combinatorial explosion in the number of execution paths. The all-paths criterion is therefore often relaxed to impose coverage of only those paths containing numbers of iterations within a user-defined limit, k. In our example, k is set to 2 so only the feasible paths containing combinations of 0, 1 or 2 loop iterations are covered. In order to implement this k-path criterion, we have extended the instrumentation of the source code so as to annotate the conditions which determine loop entry or exit. Our constraint solving strategy uses these annotations.

4 Instrumentation

The instrumentation stage is an automatic transformation of the source code so as to print out the symbolic execution path, i.e. a sequence of assignments and satisfied conditions on C variables or access paths. A trace instruction is therefore inserted after each assignment and each branch of the source code. The instrumentation is implemented using the CIL library [12]. Certain source-code statements are decomposed, notably multiple conditions which reinforces our test criterion, bringing it close to all-paths combined with MC/DC. Note that in the C language, variable values may be referenced using "data access paths" involving not only the operators to access array elements or structure fields, but also pointer de-references. In our trace instructions, all data access paths are rewritten, in a purely syntactic transformation, to a canonical form, so as to simplify the substitution stage.

5 Substitution

A path predicate is a conjunction of constraints expressed in terms of the values (at input) of the input variables. However, the symbolic conditions output by the instrumentation of the conditional statements in the source code may be expressed in terms of local variables (or intermediate values of input variables). The substitution stage of our approach carries out the projection of these conditions onto the values of the inputs. The sequence of statements output by the execution of the instrumented program is traversed and each assignment is used to update a "memory map" which stores the current symbolic value of local variables in terms of the input values. When a condition is encountered, all occurrences of local variables are replaced by their current symbolic values. The resulting list of conditions is the path predicate. Because we analyse a single, unrolled, path, we do not need to use the SSA form used in [2] and can treat aliases (two or more ways of denoting the same memory location) with relative ease.

6 Test Selection and Constraint Solving

The first test-case t_1 is generated from a selection domain SD_0 which is the input domain, ID, of the program under test. ¿From the execution of t_1, we derive the corresponding path predicate PP_1. In order to cover a new path, we have to generate test inputs from the difference, SD_1, of SD_0 and the domain of PP_1 (see Fig. 3). If SD_1 is empty, this means that there are no more paths to cover. Otherwise, we can generate a new test-case t_2, from SD_1, which exercises a new path whose predicate is PP_2. This process is repeated until an empty selection domain SD_n is reached, in which case we have covered every feasible path of the program under test.

Each path predicate PP_i is the ordered conjunction of the number p_i of successive conditions $C_{i,j}$ encountered along the corresponding path:

$$PP_i = C_{i,1} \land \ldots \land C_{i,pi} \tag{1}$$

The negation of PP_i is just the disjunction of all the prefixes of PP_i with the last condition negated :

$$\neg\, PP_i = \neg\, C_{i,1} \lor \bigvee_{m=2.....pi} (C_{i,1} \land \ldots \land C_{i,m-1} \land \neg\, C_{i,m}) \tag{2}$$

Note that each term of this disjunction is a conjunction of conditions corresponding to a (possibly infeasible) path prefix which is unexplored at the ith step of our selection strategy. To find a solution in each selection domain SD_i, we choose to solve the longest feasible conjunction in $\neg\, PP_i$, which we call $MaxC_i$. If all the conjunctions in $\neg\, PP_i$ are infeasible, the longest unsolved feasible conjunction in $\neg\, PP_{i-1}$, $MaxC_{i-1}$, is tried, and so on. Our strategy corresponds in this sense to a depth-first construction of the tree of feasible execution paths, as we will illustrate below on our example function.

To respect the k-paths criterion, the definition of $MaxC_i$ must be modified to take into account the annotations of conditions from the heads of loops with a variable number of iterations. If the negation of a condition would result in loop re-entry after k or more iterations, then it is not explored. This way, we ensure

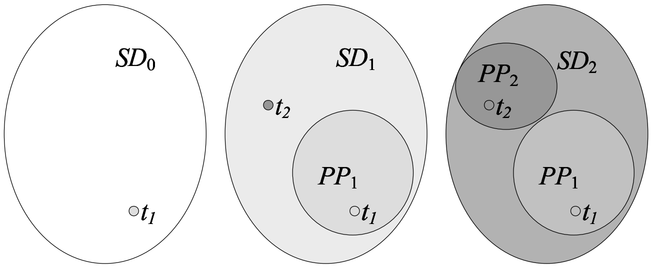


Fig. 3. Input domains

that we never generate any new path predicate prefixes containing too many loop iterations. However, we cannot prevent constraint solving of some path predicate prefix occasionally resulting in a "superfluous" test, i.e. one covering a path which - after the prefix - executes more than k iterations of a loop.

Test selection and constraint solving are implemented in the Eclipse constraint logic programming environment [17]. Note that solving non-linear constraints is decidable only for data types with finite domains, such as integers. However, current research [10, 15] holds the promise of decidable and precise constraint solving for floating-point numbers too. Solving constraints over finite domains is NP-complete in the worst case but we base our work on heuristics developed for test-case generation problems [3, 7] which display low complexity in practice. In the case of data-structures whose size may not be the same in all the test cases, constrained variables representing the elements of the data-structure are defined only as needed. Our "labelling" heuristic (used to generate and test values after constraint propagation) is to choose dimension values as low as possible. This has the advantage that we are sure to generate tests for empty data-structures (where they are allowed), whose treatment is often a source of bugs. Moreover, as there is often a link between data-structure dimensions and the number of loop iterations, smaller data-structures can result in fewer superfluous test cases for the k-path criterion. For variables other than dimensions, labelling uses a random generator which starts by generating values in the median third of the variable's domain after constraint propagation. If all these values have been tried without success, randomly generated values outside the median third are tried.

An advantage of our test generation strategy is that we only analyse feasible path predicates. Of course during the search for $MaxC_i$, we may construct other path predicate prefixes which turn out to be unsatisfiable, but this is always due to the negation of the last condition. This kind of unsatisfiability is easier to detect than that due to the structural construction of arbitrary path predicates. Moreover, when a path predicate prefix has no solution, the strategy does not construct or explore any path predicates starting with this prefix.

7 Example

Now let us follow step by step what happens when we run the PathCrawler on our example. The first step is to define the input domain, ID, of `Merge`. Note that the formal parameters of a C function may not all be input parameters and that some global variables may also be input parameters. The parameters may be accessed via pointers or belong to structured data types of possibly unknown dimensions. In our example, `t3` is not an input parameter and the sizes of `t1` and `t2` are variable. PathCrawler can treat functions with pointers and structured data, including arrays with variable dimensions, as input parameters. However, as the input parameters of a C function cannot be automatically identified without static analysis, the user is currently asked to pick out the input parameters from the list of all the scalar formal parameters and global variables visible to

dim(t1) ∈ 0 . . . 10000
l1 ∈ 0 . . . 10000
$forall$ i ∈ 0. . . dim(t1). t1[i] ∈ -100 . . . 100
dim(t1) = l1
$forall$ i ∈ 1. . .l1. t1[i] ≥ t1[i - 1]
and similarly for t2

Fig. 4. Merge domains and preconditions

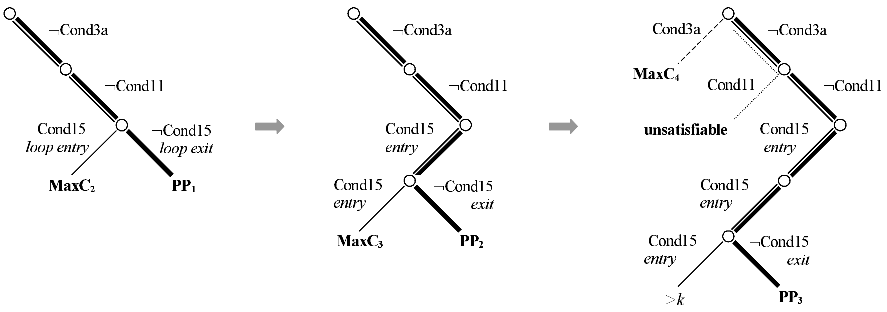


Fig. 5. Selection strategy

the function, and of all the components (elements, fields, de-referenced values,) of structured formal parameters and visible global variables, or those in the form of pointers. The user is also asked to replace the default domain (based on its declared C type) of each input parameter by a smaller interval, when applicable. Similarly, the user must give the upper limit of any variable dimensions of arrays containing input parameters. Finally the user must define any other input parameter dependencies (precondition). The $forall$ operator, which iterates over all elements of an array, can currently be used in the precondition definitions and we are studying the use of a richer language to specify the precondition. In our example (see Fig. 4), the value of `l1` (resp. `l2`) must be less than or equal to the length of `t1` (resp. `t2`) (and in fact, we set them as equal). Furthermore, `t1` and `t2` must be ordered.

In the first test-case of our example, generated from the domains and constraints of Fig. 4, the sizes of `t1` and `t2` are set to zero. This test-case is shown in Table 1, in which the arcs of the execution path are denoted by the line-number of the corresponding condition in the source code (in Fig. 1), preceded by a minus sign if the condition is not satisfied and, in the case of composite conditions, followed by a letter indicating the sub-condition concerned. The predicate PP_1 of the path covered by the first test encounters the following conditions (also numbered according to their origin in the source code), shown in Fig. 5:

$C_{1,1} = \neg\ Cond3a : \neg\ 0 < l1$ (exit 1st loop after 0 iterations)
$C_{1,2} = \neg\ Cond11 : \neg\ 0 < l1$ (exit 2nd loop after 0 iterations)
$C_{1,3} = \neg\ Cond15 : \neg\ 0 < l2$ (exit 3rd loop after 0 iterations)

Table 1. Tests generated for `Merge`

no.	l1	l2	t1[0]	t1[1]	t1[2]	t2[0]	t2[1]	t2[2]	path covered (with selected prefix underlined)
1	0	0							-3a,-11,-15
2	0	1				-3			$\underline{\text{-3a,-11, 15}}$,-15
3	0	2				-52	30		$\underline{\text{-3a,-11, 15, 15}}$,-15
4	1	0	-5						$\underline{\text{3a}}$,-3b, 11,-11,-15
5	2	0	-41	-8					$\underline{\text{3a,-3b, 11, 11}}$,-11,-15
6	1	1	-17			16			$\underline{\text{3a, 3b}}$, 4,-3a,-11, 15,-15
7	1	2	24			67	88		$\underline{\text{3a, 3b, 4,-3a,-11, 15, 15}}$,-15
8	2	1	-67	14		-22			$\underline{\text{3a, 3b, 4, 3a}}$, 3b,-4, 3a,-3b, 11,-11,-15
9	3	1	-77	-27	0	-61			$\underline{\text{3a, 3b, 4, 3a, 3b,-4, 3a,-3b, 11, 11}}$,-11,-15
10	2	1	-1	23		46			$\underline{\text{3a, 3b, 4, 3a, 3b, 4}}$,-3a,-11, 15,-15
11	2	2	-68	-37		-14	29		$\underline{\text{3a, 3b, 4, 3a, 3b, 4,-3a,-11, 15, 15}}$,-15
12	3	1	-69	-36	28	-5			$\underline{\text{3a, 3b, 4, 3a, 3b, 4, 3a}}$, 3b,-4, 3a,-3b, 11,-11,-15
13	1	1	-23			-50			$\underline{\text{3a, 3b,-4}}$, 3a,-3b, 11,-11,-15
14	2	1	41	73		9			$\underline{\text{3a, 3b,-4, 3a,-3b, 11, 11}}$,-11,-15
15	1	2	-30			-69	24		$\underline{\text{3a, 3b,-4, 3a, 3b}}$, 4,-3a,-11, 15,-15
16	1	3	-30			-73	-13	15	$\underline{\text{3a, 3b,-4, 3a, 3b, 4,-3a,-11, 15, 15}}$,-15
17	2	2	31	56		-17	64		$\underline{\text{3a, 3b,-4, 3a, 3b, 4, 3a}}$, 3b, 4,-3a,-11, 15,-15
18	1	2	27			-54	-26		$\underline{\text{3a, 3b,-4, 3a, 3b,-4}}$, 3a,-3b, 11,-11,-15
19	2	2	-52	-26		-79	-65		$\underline{\text{3a, 3b,-4, 3a, 3b,-4, 3a,-3b, 11, 11}}$,-11,-15

Solution of $MaxC_2 = \neg\ Cond3a \wedge \neg\ Cond11 \wedge Cond15$ generates the second test-case shown in Table 1, in which there is one iteration of the third loop. The third test, covering two iterations of the third loop, is generated in a similar way. With no limit on loop iterations, $MaxC_4$ would be:

$$\begin{array}{rlrl}
C_{3,1} = & \neg\ Cond3a : & \neg\ 0 < l1 & \text{(exit 1st loop after 0 iterations)} \\
C_{3,2} = & \neg\ Cond11 : & \neg\ 0 < l1 & \text{(exit 2nd loop after 0 iterations)} \\
C_{3,3} = & Cond15 : & 0 < l2 & \text{(entry 1st iteration of 3rd loop)} \\
C_{3,4} = & Cond15 : & 1 < l2 & \text{(entry 2nd iteration of 3rd loop)} \\
\neg\ C_{3,5} = & Cond15 : & 2 < l2 & \text{(entry 3rd iteration of 3rd loop)}
\end{array}$$

This is where the modification of our strategy to limit loop iterations takes effect: this conjunction is not solved because it would entail more than 2 iterations of the third loop. Our strategy thus backtracks to the lowest unexplored branch of Fig. 5 and constructs the path prefix $\neg\ Cond3a \wedge Cond11$. However, this is unsatisfiable, so $MaxC_4$ is in fact $Cond3a$.

Of the 19 tests generated in our example, only the 12th and 17th are superfluous (contain more than 2 iterations of the same loop) and we only need to discover the infeasibility of 25 path predicate prefixes. In comparison, `Merge`'s control-flow graph contains 109 infeasible paths if k is set to 2.

To test the efficiency and stability of our implementation, we ran our prototype ten times on `Merge` with k set to 5 and maximal domains for the elements of `t1` and `t2`. For this value of k, the control-flow graph contains 4536 paths, of which 321 are feasible. In each run, 337 tests were generated and 317 infeasible path predicate prefixes found in order to eliminate the 4215 infeasible paths, i.e. 654 predicate prefixes were generated and solved or rejected. The CPU execution time on a 2GHz PC running under Linux varied between 0.75 and 0.81

seconds, with an average of 0.785. In conclusion, all k-paths were tested (and 20 superfluous tests generated) in under 1 second and our random labelling heuristic did not cause much variation in execution time. For $k = 10$, 20993 tests are generated and 15357 infeasible paths eliminated in around 116 seconds.

8 Other Examples

We have also tried our prototype on some well-known examples from the testing literature: the programs TriType, Bsort and Sample (see Appendix). Given the lengths of the sides of a triangle, Tritype carries out a series of tests on them to classify the triangle. It has no loops and only 14 execution paths but is interesting because the path predicates include simple arithmetical expressions and not just inequalities as in the other examples. Bsort is a bubble sort containing two nested loops, one iterating over all the elements of the array to be sorted and the other over the elements after the current one. This example demonstrates the limits of our current implementation of the k-paths strategy : the number of superfluous tests grows exponentially with k due to PathCrawler's attempts to find paths with k executions of both loops. The best way to limit the number of paths in this case is therefore by reducing the length of the array to be sorted. Sample compares the contents of two arrays to a reference value in two successive loops, each with a fixed number of iterations of the length of the array. For array lengths n and m, the number of paths is 1+ (2^n -1) * 2^m. The k-path strategy cannot be used for this example because the number of iterations is fixed but the number of paths can be kept reasonable by limiting n and m, which is justified by the total lack of dependence between successive loop iterations. The number of tests, number of infeasible prefixes, mean execution time in seconds and variation in the execution times over 10 runs are shown in Table 2

Table 2. Results for other examples

program	k	array dimn.	tests	infeasible prefixes	mean exec. time	min exec. time	max exec. time
TriType	-	-	14	3	0.01	0.01	0.02
Bsort	10000	0 - 5	153	349	1.16	1.14	1.17
Sample	-	4	241	0	0.27	0.22	0.29

9 Further Work

Our first priority is to apply PathCrawler to a wide range of larger examples. However, our results so far suggest that our approach is efficient enough to scale up to the treatment of larger programs providing that we limit the combinatorial explosion of the number of execution paths. We have shown how we easily adapted our test selection strategy to limit the number of iterations of certain loops. Our current topics of investigation [11] include strategies to avoid testing all the paths in each call to another function. Our method is open to such modulations in the test strategy. Firstly, constraints other than those from a path predicate can be taken into account, as is already done for the treatment of the precondition on the input values of the program under test. Integration test scenarios could be used in the same way. Secondly, information collected during execution of the program under test can also influence test selection, as illustrated by the use of annotations of loop-head conditions to implement the k-path criterion. By annotating the conditions in called functions, the exploration of different paths in these functions could be restricted.

However, the effectiveness of our test generation strategy is limited by the selection of a single test for each path. It could be easily modified to select tests at the limits of the path domain boundaries [4], where bugs are often found. The chances of detecting coincidental correctness would be improved if we also extended our random generation of variable values to the random generation of several tests for each path, in a similar way to statistical structural testing [16, 3].

Finally, the applicability of PathCrawler depends on a high degree of automation of the test process. In the current prototype, the oracle must be hand-coded but by taking certain forms of post-condition on the C variables into account, we could automatically generate the oracle as in [6, 13].

References

1. M.J. Gallagher and V.L. Narasimhan, ADTEST : A Test Data Generation Suite for Ada Software Systems, IEEE Transactions on Software Engineering, Vol. 23, No. 8, August 1997
2. A. Gotlieb, B. Botella and M. Reuher, A CLP Framework for Computing Structural Test Data, CL2000, LNAI 1891, Springer Verlag, July 2000, pp 399-413
3. S-D Gouraud, A. Denise, M-C. Gaudel and B. Marre, A New Way of Automating Statistical Testing Methods, ASE 2001, Coronado Island, California, November 2001
4. B. Jeng and E.J. Weyuker, A Simplified Domain-Testing Strategy, ACM Transactions on Software Engineering and Methodology, Vol. 3, No. 3, July 1994, pp 254-270
5. B. Korel, Automated Software Test Data Generation, IEEE Transactions on Software Engineering, Vol. 16, No. 8, August 1990
6. G.T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D.R. Cok, How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification, In Formal Methods for Components and Objects, LNCS Vol. 2852, Springer Verlag, Berlin, 2003, pp 262-284
7. B. Marre and A. Arnould, Test sequences generation from Lustre descriptions: GATeL, ASE 2000, Grenoble, pp 229–237, Sep. 2000
8. B. Marre, P. Mouy and N. Williams, On-the-Fly Generation of K-Path Tests for C Functions, 19th IEEE Intnl. Conf. on Automated Software Engineering (ASE 2004), September 2004, Linz, Austria.
9. C. Michael and G. McGraw, Automated Software Test Data Generation for Complex Programs, ASE, Oct 1998, Honolulu

10. C. Michel, M. Rueher and Y. Lebbah, Solving Constraints over Floating-Point Numbers, CP'2001, LNCS vol. 2239, pp 524-538, Springer Verlag, Berlin, 2001
11. P. Mouy, Vers une méthode de génération de tests boîte grise "à la volée", Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'04), June 2004, Besançon, France
12. G.C. Necula, S. McPeak, S.P. Rahul and W. Weimer, CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs, Proc. Conference on Compiler Construction, 2002.
13. M. Obayashi, H. Kubota, S.P. McCarron and L. Mallet, The Assertion Based Testing Tool for OOP: ADL2, In Proc. ICSE'98, Kyoto, Japan, 1998
14. R.E. Prather and J.P. Myers, The Path Prefix Testing Strategy, IEEE Transactions on Software Engineering, Vol. 13, No. 7, July 1987
15. N.T. Sy and Y. Deville, Consistency Techniques for Interprocedural Test Data Generation, ESEC/FSE'03, September 1-5, 2003, Helsinki, Finland
16. P.Thevenod-Fosse and H.Waeselynck, Software statistical testing based on structural and functional criteria, 11th International Software Quality Week, San Francisco (USA), 26-29 May 1998
17. M. Wallace, S. Novello and J. Schimpf, ECLiPSe: A Platform for Constraint Logic Programming, IC-Parc, Imperial College, London, August 1997

Appendix

```
int tritype(int i, int j, int k){
  int trityp;
  if ((i == 0) || (j == 0) || (k == 0)) trityp = 4;
  else {
    trityp = 0;
    if (i == j) trityp = trityp + 1;
    if (i == k) trityp = trityp + 2;
    if (j == k) trityp = trityp + 3;
    if (trityp == 0){
      if ((i+j <= k) || (j+k <= i) || (i+k <= j)) trityp = 4;
      else trityp = 1;
    }
    else if (trityp > 3) trityp = 3;
    else if ((trityp == 1) && (i+j > k)) trityp = 2;
    else if ((trityp == 2) && (i+k > j)) trityp = 2;
    else if ((trityp == 3) && (j+k > i)) trityp = 2;
    else trityp = 4;
  }
  return trityp;
}


void bsort (int * tableau, int l)
{
  int i, temp, nb;
  char fini;
  fini = 0;
  nb = 0;
  while ( !fini && (nb < l-1)){
    fini = 1;
    for (i=0 ; i<l-1 ; i++)
      if (tableau[i] < tableau[i+1]){
        fini = 0;
        temp = tableau[i];
        tableau[i] = tableau[i + 1];
        tableau[i + 1] = temp;
      }
    nb++;
  }
}


int sample(int a[4], int b[4], int target)
{
  int i, fa, fb;
  i=0;
  fa=0;
  fb=0;
  while(i<=3){
    if(a[i]==target) fa=1;
    ++i;
  };
  if(fa==1){
    i=0;
    fb=1;
    while(i<=3){
      if(b[i]!=target) fb=0;
      ++i;
    }
  }
  if(fb==1) return 0;
  else return 1;
}
```

A New Methodology and Tool Set to Execute Software Test on Real-Time Safety-Critical Systems

Caiazza Alessandro, Di Maio Roberto, Scalabrini Fernando, Poli Fabio, Impagliazzo Leonardo, and Amendola Arturo

ANSALDO SIGNAL - Ansaldo Segnalamento Ferroviario S.p.A.
Via Nuova delle Brecce, 260
80147 Napoli, Italy
Phone: +39.081.243.2981 – Fax: +39.081.243.7089
{caiazza.alessandro, dimaio.roberto, scalabrini.fernando, poli.fabio, impagliazzo.leonardo, amendola.arturo}@asf.ansaldo.it

Abstract. CENELEC norms identify four testing phases for the verification and validation of real-time safety-critical software for railway control and protection systems (Module, Integration, Hardware/Software Integration and System testing). The RAMS team of Ansaldo Segnalamento Ferroviario (ASF) designed a methodology that allows executing simultaneously these four phases, also allowing performing code coverage measurements. Several proprietary tools are needed to apply this methodology to perform test directly on the target system and to record coverage measures during normal operation, with negligible intrusion in system performances. The commercial tools do not allow test execution on real prototypes without affecting performances. The proposed tool set will aid the test engineer from the test specification to the results checking, including the test documentation, regression analysis and reports archiving. By using this environment, the application of the methodology will be optimized, and the verification and validation process will be managed in a harmonized and standardized way.

1 Introduction

Recent years have seen the demand for a huge increase in the reliability and performance of control systems in railway and metro lines. This demand has required the transition from relay to computer-based systems, stressing the need for the design and assessment of the safety of completely new systems. The safety of railway systems is based on the fail-safe behavior of their components. This concept is well assessed for relays, which are characterized by defined failure modes, but it is hardly applicable to modern computer-based systems.

CENELEC norms [1, 2, 3], approved as European standards, require the use of verification and validation processes in all phases of the life cycle of the system. These recommendations coupled with the availability of microprocessor devices and with strong competition in the railway field have brought about the definition and development of new methodologies for safety design and assessment.

CENELEC norms identify four testing phases for the verification and validation of real-time safety-critical software for railway control and protection systems: Module Testing, Integration Testing, Hardware/Software Integration Testing, and System Testing.

In recent past, the RAMS team of Ansaldo Segnalamento Ferroviario designed a methodology that allows executing simultaneously these four phases, also allowing performing code coverage measurements. This methodology was successfully applied in the verification and validation of newer microprocessor based systems realized by Ansaldo Segnalamento Ferroviario (ACS Roma-Termini interlocking, SCMT ATP) and US&S (Copenhagen driverless Metro), and was approved by several assessors. The introduction of this methodology allowed a remarkable reduction of the verification and validation costs.

The methodology identified and adopted by the RAMS team of Ansaldo Segnalamento Ferroviario relies on proprietary tools, as available commercial tools do not allow test execution on real prototypes without affecting performances: i.e., with the low level of intrusion required by Real-Time Safety-Critical Systems.

CENELEC norms require exhaustive documentation of all the phases of the V&V software process, to allow an external assessor to precisely understand completeness and consistence of the verification and validation activities, and eventually to rerun some of the tests in order to ascertain the repeatability of their results.

The RAMS team of Ansaldo Segnalamento Ferroviario developed a tool set to standardize and automate the various phases of the V&V software process and to automatically generate the documentation requested by the norms.

This paper aims to describe the methodology adopted by RAMS group of Ansaldo Segnalamento Ferroviario and relevant proprietary tools implemented to apply this methodology and to optimize several phases of the V&V process. The work is organized as follows: Section 2 presents the methodology adopted by RAMS group of ASF (Description of methodology), in Section 3 it is described the tool used to monitor software I/O variables (Monitoring of software I/O variables and coverage measurement); in Section 4 it is described the tool used to manage the automation and standardization of V&V process (Manage code instrumentation and documentation). Section 5 presents the Conclusions and future works.

2 Description of the Methodology

CENELEC norms do not specify what techniques must be used to test real-time safety-critical software, but the methodology, the tools, the results and more generally the entire design and verification and validation process have to be approved by an independent safety Assessor.

Different methodologies can be used to execute different test categories; in particular the methodology described in this paper is applied to execute functional tests and coverage measures on hard real time systems (low intrusion is required) [5].

By using the methodology adopted by the RAMS group of ASF, the tests required by the CENELEC norms are executed directly on the target system.

The V&V process is executed after the software development phase, therefore an environment simulator and a prototype of the target system, as well as stable software version, are normally available.

In order to execute software integration testing and to get coverage measures on the functions (debug testing was executed in development phase) it is required to monitor interfaces of functions and to trace the executed paths.

The aim of ASF methodology is to perform module testing, software integration testing, hardware/software integration testing and system testing at the same time. To make this possible it is required to monitor the interfaces of software functions and to measure the code coverage on the target system, without affecting performances, while the environment simulator runs the system test scenarios.

To ensure the required low intrusion, several proprietary tools are needed.

Available commercial tools (e.g., CANTATA) perform at best offline execution of module and integration tests, but drivers and stubs are required to test the code slice. Moreover when execution of the tests on target system is supported by commercial tool, it is affected by high level of intrusion (tracing data are generally sent through serial ports) making it unusable in hard real time context. Our methodology and related tools work at best on target systems and the obtained level of intrusion allows the respect of hard real time constrains. They are not suitable only to monitor low-level functions (e.g., interrupt handlers) and to get timing measures. Logic Analyzer is recommended in those cases, that impact on 10% of testing at most [4], [5].

3 Monitoring Software I/O Variables and Coverage Measurements

As anticipated in previous section, even if several off-the-shelf tools are available to monitor system behavior, ASF decided to develop a proprietary tool set to overcome their limitations.

In fact, off-the-shelf testing environments can be distinguished in two main categories:

- Hardware-level testing environments (e.g., Logic Analyzer, debug monitor, JTAG/BScan);
- Software-level testing environments (e.g., CANTATA, LogiScope).

Hardware-level testing environments guarantee a very low intrusion, but it is generally hard to trace low-level information provided by these tools with the high-level ones (e.g., software interfaces behavior) needed during software/system V&V activities. As a consequence of this a very small amount of test is executable with those environments. When needed (e.g., to monitor low level drivers or interrupt handlers), ASF prefers using Logic Analyzers [4], as others environments such as debug monitors usually use a set of procedures in order to inhibit (partially or totally) the use of some resources (e.g., MMU, interrupt handler) by the software under test [6].

Software-level testing environments usually run in an emulated environment on a workstation, so time evaluations are not accurate and the simulation of Input/Output resources behavior is imprecise. Obviously such environments cannot be used to test real-time systems (ASF uses them for module level debugging, during the design phase). Some of these environments (e.g., CANTATA) also allow performing the tests on the target system. Anyway Stubs and Drivers usually need to be designed to execute tests on a set of functions. Stubs and drivers simulate the system “surrounding” the tested functions, in this way only integration test is executed on the functions under test. This is not needed when a prototype the whole system is available. Furthermore interfaces visibility often is limited to the functions under test. Finally ASF usually encountered problems in letting such environments run when the complexity of the system increases.

As a consequence of these limitations, ASF decided to design proprietary tools capable to work in hard-real-time contexts (several optimizations were realized to minimize time execution of monitoring task) and to limit the source code instrumentation just on the functions under test, as described below.

To verify that the software satisfies its requirement specifications (at any level: module or integration), it is required to monitor the function Input/Output parameters and global variables while executing software tests. The tool IntMon (Interface Monitoring) supports such task.

IntMon instruments the code, with a negligible intrusion level, to store Input/Output data of the functions under test in system RAM.

The tool automatically generates the functions to print in RAM the input data (MyFuncPInput ()) and output data (MyFuncPOutput ()). It needs information about name, type and typology (input, output or input-output) of parameters and global variables used by the function under test. Such data are expected in function headers (code comment before function definition) as required by our coding standard.

During the test execution, this sequence is followed:

1. print of the input variables (call of the function MyFuncPInput ());
2. execution of the function under test (call of the function MyFunc ());
3. print of the output variables (call of the function MyFuncPOutput ());

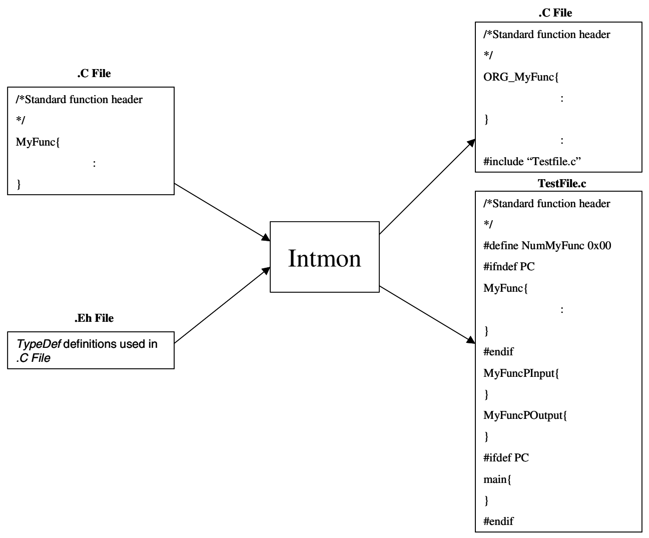


Fig. 1

In order to force this sequence the source code is slightly modified: the function under test (MyFunc()) is renamed adding "ORG_" to its name (ORG_MyFunc()) (see fig. 1). A new file is added to the project (TestMyFunc.c) and included at the end of the .c file; TestMyFunc.c contains a new version of MyFunc() and the printing functions (see fig. 1).

During the normal execution of the software on target system, the new function MyFunc() is called instead of the original ORG_MyFunc(). The new MyFunc() forces the execution sequence described before calling the three functions: MyFuncPInput(), ORG_MyFunc() and MyFuncPOutput().

The new MyFunc() function needs some changes in order to execute tests (before the development of the tool CIRO described in the following section those changes were executed manually by the test engineer):

- The function under test must be identified univocally therefore a symbolic constant NUM_MyFunc is defined in the testMyFunc.c file with default value equal to zero. The test engineer has to modify this value ensuring a unique value for each function.
- A global variable (testNmbrVV) is used to activate the monitoring of the interfaces of the function. The source code of MyFunc() contains a switch statement to enable (flagTest=1) or disable (flagTest=0) the monitoring, depending on the value of TestNmbrVV. The test engineer has to edit the template cases produced by the tool as appropriate (see examples of tuned switch statements in Figures 6 and 7).

As examples of execution sequence consider the following original code where function func1() in the file1.c file calls function func2() contained in the file2.c file:

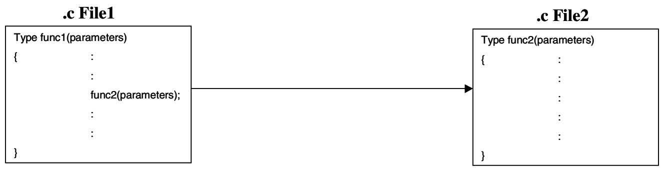


Fig. 2

After tool execution, on the instrumented func2 it is possible to perform or not the monitoring, with respect to the value of flagTest. The following cases can be identified:

1. If flagTest = 0 we get the following execution sequence:

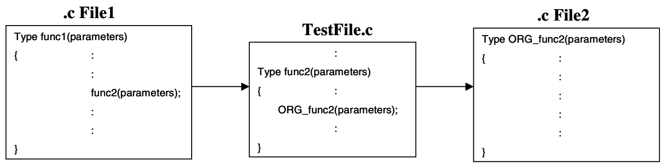


Fig. 3

2. If flagTest = 1 we get the following execution sequence:

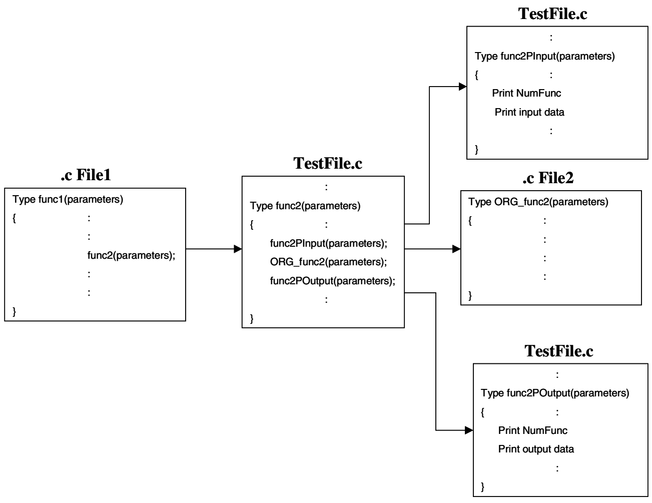


Fig. 4

The TestMyFunc.c file produced by IntMon is compiled and linked to the original software. The instrumented code is loaded and executed directly on target system (fig.5).

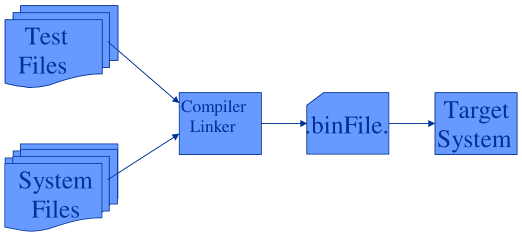


Fig. 5

To set the value of testNmbrVV a command is sent via serial port. During the execution of the test the functions MyFuncPInput() and MyFuncPOutput() write, in a reserved RAM area on target system, interface data of the function under test. At the end of the test, those values are downloaded via serial port on the PC and stored in a file (report file).

The data download is executed at the end of the test, the system isn't operative and it is possible to use the serial port to download data, even if it was used as system resource during the operative phase.

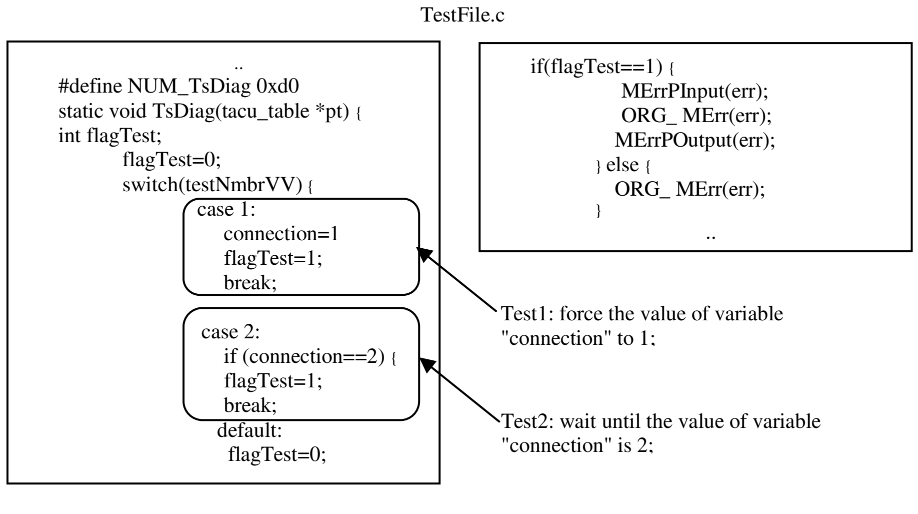
The test engineer can build several tests using IntMon, compiling and loading the code on target system just once. In fact, it is possible to add various branches to the switch contained in the function MyFunc() produced by the IntMon. During execution phase, the test engineer executes the various tests planned by setting, for each test, the appropriate value of the variable testNmbrVV.

In the case statement contained in the function MyFunc() it is possible to insert some trigger conditions to enable the monitoring of the interfaces only when specified events happen or force the value of one or more variables (fig.6). Test engineer executes those changes with the aid of the tool CIRO described in the following section.

Forcing the values of some variables during tests can be done:

1) To simulate faults (software fault-injection) in order to verify the behavior of the system (error condition detected or not).

2) To simulate two synchronous external events (when it is very hard to create the real situation by acting on the boundaries of the whole system): When the first event happen the second one is simulated by software fault-injection.


TestFile.c
Test1: force the value of variable "connection" to 1;
Test2: wait until the value of variable "connection" is 2;

```
                  ..
#define NUM_TsDiag 0xd0
static void TsDiag(tacu_table *pt) {
int flagTest;
        flagTest=0;
        switch(testNmbrVV) {
                case 1:
                   connection=1
                   flagTest=1;
                   break;
                case 2:
                   if (connection==2) {
                   flagTest=1;
                   break;
                 default:
                  flagTest=0;
```

```
if(flagTest==1) {
          MErrPInput(err);
          ORG_ MErr(err);
         MErrPOutput(err);
        } else {
         ORG_ MErr(err);
        }
                   ..
```

Fig. 6

Several functions can be observed in the same test. For each function, a test file (TestMy_X_Func.c) containing the functions My_X_Func() and the two functions to store the Input/Output variables will be produced. The test engineer has to insert the same case statement contained in the files produced by the tool to activate the monitoring of all functions under test (those operations are completely automated by using the tool CIRO described in the following section).

The sequence of data in the report file corresponds exactly to the activation sequence of the functions under test. If a function func1() calls a function func2(), in the report the data will be presented in the following order:

input func1() => input func2() => output func2() => output func1().

In some cases it is necessary to monitor function func2(), called by several functions, only when it is called by a specific function func1(). In this case the test engineer has to modify (with the aid of the tool CIRO described in the following section) the switch contained in the test files related to the functions under test in order to create a "chain" as shown in (fig.7).

```
                  ..
#define NUM_TsDiag 0xd0
static void TsDiag(tacu_table *pt) {
int flagTest;
        flagTest=0;
        switch(testNmbrVV) {
                case 1:
                   flagTest=1;
                   testNmbrVV=2;
                   break;
                default:
                  flagTest=0;
        }
        if(flagTest==1) {
            TsDiagPInput(pt);
            ORG_TsDiag(pt);
           TsDiagPOutput(pt);
        } else {
           ORG_TsDiag(pt);
        }
}
                  ..
```

```
                  ..
#define NUM_MErr 0xe0
static void MErr(int err) {
int flagTest;
        flagTest=0;
        switch(testNmbrVV) {
                case 2:
                   flagTest=1;
                   testNmbrVV=0;
                   break;
                default:
                  flagTest=0;
        }
        if(flagTest==1) {
            MErrPInput(err);
            ORG_ MErr(err);
           MErrPOutput(err);
        } else {
           ORG_ MErr(err);
        }
}
                  ..
```

Fig. 7

The test engineer can obviously realize more complex "chains" to involve several functions.

IntMon makes use of a commercial tool, called LogicScope, to get coverage measurement. LogicScope normally works on PC changing the function under test in order to store in a file the list of branches covered during the execution of the code. To get coverage measures on target system the LogicScope libraries have been substituted by proprietary ones in order to store data in a reserved RAM area. IntMon instruments the code in such a way to minimize intrusion (also allowing to limit coverage measures to a single function within a module), as LogiScope instrumentation is incompatible with real-time systems. After execution of the test, IntMon generates the coverage data in a format compatible with LogiScope, so to allow use of graphic analysis capabilities provided by the commercial tool.

IntMon allows coverage measures on the target system by calling the function created offline by LogicScope (COV_MyFunc()) instead of the original function (ORG_MyFunc()).

To execute coverage measurements on the target system, the test engineer has to:

- Execute LogicScope to create COV_MyFunc() on PC.
- Execute IntMon to create test file as described above (COV_ instead of ORG_).
- Compile the instrumented code and load it on target system.
- Execute the test.

Coverage measures and interface data are stored in independent RAM areas of the target system.

When coverage is measured the intrusion can highly increase depending on the number of instrumented functions. For complex systems it is almost impossible to measure coverage on the whole software, so a partitioning of the system software is needed in order not to impact too much on real-time properties.

4 Managing Code Instrumentation and Documentation

In order to standardize and automate the various phases of the software V&V process an additional tool was implemented. It calls IntMon and supports the activity of the test engineer by managing the instrumentation of the code and the generation of the documentation requested by the international norms for the V&V activity.

Through the tool IntMon it is possible to monitor the variables representing the software interfaces as well as the code branches exercised during the execution of the test cases.

Before the development of the tool described in this section, a series of manual and not standardized operations had to be performed. First of all, in accordance with the scope of each designed test case, it was necessary to modify the code under test by manually inserting the probes for the interfaces monitoring and the code coverage recording. Specific instructions could be used to "trigger" the acquisition of information when specific operational conditions happen. Then, after execution of the test on the target system, the recorded data had to be manually compared with the expected ones.

All these operation have to be exhaustively documented.

The tool CIRO (Code Instrumentation & Report Organizer) is a PC based environment able to manage the set of operations needed to prepare the tests. The tool CIRO aids the test engineer from the test specification to the results checking, including the test documentation and the reports archiving.

By using this environment, the application of the methodology described in section 2 is optimized, and several additional functionalities are introduced to manage the verification and validation process in a harmonized and standardized way.

First of all, the tool parses the source code in order to recover information on all interfaces and symbols in the software under test. The test engineer will specify the test cases directly in this computer-based environment, with a reduced effort and minimizing the risk to commit errors.

The tool records in a database all the test specifications generated by the test engineer to validate the code against its requirements and – automatically and in a standardized way – performs on the software under test all needed modifications (to monitor the interfaces dealing with each test case, to record the code paths covered during the test execution, and to "trigger" all this data storing just when it is the case, in accordance with the scope of the test case).

After execution of the test cases on the target system, the test engineer will load the recorded data in the tool, which will automatically compare the test results with the expected ones. This will lighten the test engineer from the task to manually compare tenths of values for every test case, also getting rid of any chance of human mistakes.

The availability of all information in the database allows the tool to automatically produce the test specifications documents (hundreds of pages for a medium size software component), to ensure complete traceability for every test case (specification, setup, execution logs and result), as well as to produce detailed statistics on the test set (exercised functions, monitored variables, and so on).

As complex software projects, such as the ones designed for railway control and protection systems, always go through a certain number of reviews (also due to the results of the verification and validation activities), it is very important that the tool is able to help the test engineer in the task of performing the regression analysis and testing. By comparing the newest software revisions with respect to the previously tested ones, the tool is able to recover those tests which are not influenced by the changes, to identify the tests that have to be re-executed, and to help the test engineer in re-design tests related with parts of the code whose interfaces did change or with modified functional requirements. This can be done after a comparison of the code versions, as in the database it is possible to identify all parts of the software and all requirements involved in a certain test case. All test cases related with unchanged code and requirements can be still considered valid, also for the new software version, as far as the test engineers correctly identified and tested independent software modules.

By collecting all information for each tested software version, the tool also will improve the traceability of all verification and validation activities during the whole project lifecycle.

The automation of all the manual activities required by the described methodology allows the test engineer to save time during testing, to speed up the training of new test engineer and to increase the level of quality due to the application of a standardized process, to minimize typing errors during test specification phase, to avoid errors while comparing test outcomes with expected values, and to ensure alignment and traceability between test specifications and the code version.

Keeping test data in a database allows to make analyses on the testing activity (e.g., reduction in the number of SW bugs among new SW releases) required by some assessment process and to provide quantitative statistics on testing process as required by the application of the ISO9000:2000 and CMMI level 4.

Going into details, the following phases characterize the selected testing process:

- Test Engineer deeply analyzes software requirements and other software design documentation (and, if it is necessary, the source code);
- If it is not the first software release, he/she performs a difference and impact analysis between the previous and current version of code under test;
- He/she specifies the test against the software requirements;
- He/she prepares the test specification document as required by the CENELEC norms;

- He/she instruments the source code to record the data and coverage measurement on the target system;
- He/she executes tests on target system;
- He/she analyses and verifies the results;
- He/she prepares the test report document.

A safety critical system project, executed in the years 2002/2004, is used as case study to evaluate the effort and the benefit of a fully automated testing environment. The previously described phases required the following effort:

Table 1

V&V Phase	Men-months
Phase 1 – Analysis	23.33%
Phase 2 - Comparison	6.67%
Phase 3 – Test specification	15.56%
Phase 4 – Test specifications documentation	12.22%
Phase 5 – Code instrumentation	8.89%
Phase 6 – Test execution	16.67%
Phase 7 – Test results checking	7.78%
Phase 8 - Test reports documentation	8.88%
Total	100%

Using the tool CIRO phases 5 and 7 are totally automated and effort in phases 2, 4 and 8 is reduced by the 50 ÷ 70 % (evaluation of those reductions was obtained by comparing the effort spent in the latest project, when the tool CIRO was available, with the V&V effort spent on previously validated systems, when all activities had to be manually performed).

Table 2 refers to an hypothetical project that would require a V&V effort of 100 men-months to evaluate the time reduction using the tool CIRO:

Table 2

V&V Phase	Men-months Without CIRO	Reduction	Men-months Using CIRO
Phase 1 – Analysis	23.33		23.33
Phase 2 - Comparison	6.67	50%	3.33
Phase 3 – Test specification	15.56		15.56
Phase 4 – Test specifications documentation	12.22	75%	4.44
Phase 5 – Code instrumentation	8.89	100%	0
Phase 6 – Test execution	16.67		16.67
Phase 7 – Test results checking	7.78	100%	0
Phase 8 - Test reports documentation	8.88	62.5%	3.33
Total	100		66.66

The use of the tool allows reduction of one third in men-months required by testing activities.

5 Conclusions and Future Work

This paper describes a methodology and relevant tools used by the RAMS team of Ansaldo Segnalamento Ferroviario to test modern computer based railway control systems.The described methodology is applied to functional testing and coverage measurement of real-time safety-critical systems.

Ansaldo Segnalamento Ferroviario developed a tool, named IntMon, to monitor input and output variables, to perform software fault-injection, and to evaluate code coverage with controllable intrusion.

To support the analysis of input and output variables and to standardize the V&V process a tool called CIRO was developed. It allows the test engineer to specify the test cases (automatically producing the test plan document), to generate the instrumented code ready to be uploaded to the target system, to analyze the downloaded results, and to verify their correspondence with expected ones. CIRO calls IntMon to get instrumented code.Regression analysis is supported by the tool CIRO, allowing to identify the tests affected by software changes.

To improve the tool set (IntMon/CIRO) in order to automatically generate the required headers of the functions under test a new tool is being developed. The tool will parse the source code in order to get information on all variables and symbols used in the software under test, those information will be used by the tool to build/check the header of each function of the source code as required by our coding standard.

Acknowledgments

The authors wish to thank Marco Onotri, Marco Sgambati and Amato Fusco for their precious collaboration.

References

1. EN 50126, “The Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS) for Railways Application”.
2. EN 50128, “Railway Application: Software for Railway Control and Protection Systems”.
3. EN 50129, “Railway Application: Safety Related Railway Control and Protection Systems”
4. A. M. Amendola et Al., “Experimental Evaluation of Computer-Based Railway Control Systems”. Proceedings of FTCS-27, Seattle, June 1997.
5. L. Impagliazzo et F.Poli, “The Birth and Growth of “LIVE” - Development of an hybrid fault injection environment “,"Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation", Ed. Kluwer
6. A. M. Amendola, et Al., “Innovative techniques for analysis and experimental validation of signalling and automation systems”. Proceedings of AEI-CIFE (in Italian), Firenze, September 1996.

A Grey-Box Approach to the Functional Testing of Complex Automatic Train Protection Systems

De Nicola Giuseppe, di Tommaso Pasquale, Esposito Rosaria,
Flammini Francesco, Marmo Pietro, and Orazzo Antonio

ANSALDO SIGNAL - Ansaldo Segnalamento Ferroviario S.p.A.
Via Nuova delle Brecce 260, Naples, Italy
{denicola.giuseppe, ditommaso.pasquale,
esposito.rosaria, flammini.francesco,
marmo.pietro, orazzo.antonio}@asf.ansaldo.it

Abstract. Systematic functional testing is a fundamental step of embedded control systems development cycle, as it allows to verify and validate their final implementation. Various approaches to black-box testing have been proposed, however they either involve test-case explosion or do not ensure the correctness of system behaviour in scenarios not covered by system specifications. To cope with such issues, a methodology which better suits both complexity and safety-criticality of the target system is needed. This paper describes the ASF functional testing methodology, based on a grey-box approach aimed at generating and reducing an extensive set of influence variables and test-cases. The methodology, embracing different aspects of system test process (code coverage verification, regression testing, etc.), was successfully applied to validate ASF implementation of SCMT (an Italian project for an Automatic Train Protection System). The results obtained in our testing experience proved the time effectiveness and extensive coverage of the proposed approach.

1 Introduction

Safety-critical real-time systems require a thorough testing activity [1], regulated by international standards [2]. The verification of system implementation against its functional requirements is usually pursued by means of black-box testing approaches [4]. Partition testing [5] is the most spread functional testing technique. It consists in dividing the input domain of the target system into properly chosen subsets and selecting only a test-case for each of them. Equivalence partitioning, cause-effect graphing [3], category-partition testing [6] and classification-tree method [7] are all specializations of the partition testing technique.

Functional testing techniques are based on application specifications; when these specifications are expressed through natural language, they often prove to be incorrect, incoherent, and incomplete. Functional testing, moreover, does not allow to estimate the achievable level of accuracy. For this reason, white-box (i.e. structural) testing methods can integrate functional techniques in order to check effectiveness of the test set through the code coverage measure [8]. In this case, however, especially for complex dependable systems, it is very difficult to cover all code structures, because of the great amount of defensive programming, and identify the tests that are needed to exercise the uncovered pieces of software.

Starting from this background, Ansaldo Segnalamento Ferroviario (ASF), with the aim of performing both the verification and the validation of the SCMT[1] system, that is a complex Automatic Train Protection System (ATPS), developed the hybrid approach described in this paper. The ASF approach to functional testing had the aim to overcome the problems of traditional functional testing approaches, by integrating some of the advantages of the white box methodologies.

The approach we are going to describe, though empiric, has strong theoretical basis, and has proven to be very effective during our testing experience, achieving all the objectives we needed to pursue, that is:

- Validating system implementation against its specifications (traditional functional testing objective);
- Validating the completeness and coherence of the specifications, by selecting an extensive set of input sequences and checking corresponding outputs;
- Reducing the number of needed test-cases;
- Checking code coverage;
- Performing non regression tests on modified software versions.

We could define our methodology as a grey-box hybrid approach to functional testing, because:

- The aim is not only verifying system implementation against its specification, but also validating system behaviour against unpredicted input sequences;
- System is decomposed in order to reduce the complexity of the testing phase, and such a decomposition is validated by means of static code independence checking;
- Output checking is performed sometimes on the internal state of the system and not on its "visible" outputs;
- Code coverage analysis is used to check test effectiveness.

This paper is organised as follows: Section 2 describes the system under test, in terms of working principles and hardware architecture; Section 3 provides an in depth description of the grey-box testing approach, based on a multi-level orthogonal decomposition of the target system which, together with other test-case reduction techniques, allows to dramatically reduce the complexity of the testing phase; Section 4 contains a brief discussion about the results and the future applications of the proposed methodology.

2 Description of the System Under Test

SCMT is the name of the Italian ATPS to be used on traditional rail lines. SCMT is made up by two principal sub-systems: an on-board part, physically installed in train

[1] SCMT is the acronym of "Sistema Controllo Marcia Treno", that is the Italian for "Train Movement Control System".

cockpit, and a ground part, distributed near the rail-lines. Two different types of devices are used in the ground sub-system to interact with the on-board sub-system. The first type of ground apparels consists in the track-circuits, which are devices using rail-lines to transmit data to the on-board sub-system, allowing a semi-continuous signalling system. Track-circuits are meant to send to the on-board system the status of the signals which the train is going to reach. Such information is constant during the time the train takes to travel along the loop made up by the rail-lines (typically, 1350 meters long). The second type of device is named balise, which is a transmitting antenna energized by trains passing over it, constituting a discontinuous communication system. Balises can be static or dynamic and they are able to provide the on-board system with more data respect to track-circuits. To be able to update their data, according to track status, dynamic balises must be connected to a proper encoder system. To get the information it needs from the ground sub-system, the on-board sub-system has to be connected to the TCTM (Track Circuit Transmission Module), which receives data transmitted by track-circuits[2], and BTM (Balise Transmission Module), which energises balises and reads their messages. Finally, a Man Machine Interface (MMI) is used to allow train driver interaction with the on-board sub-system. All these information are managed by the on-board sub-system which has to ensure train safety by elaborating the allowed speed profiles (i.e. dynamic protection curves) and activating service or emergency brakes in case of dangerous situations. The described architecture is depicted in **Fig. 1**.

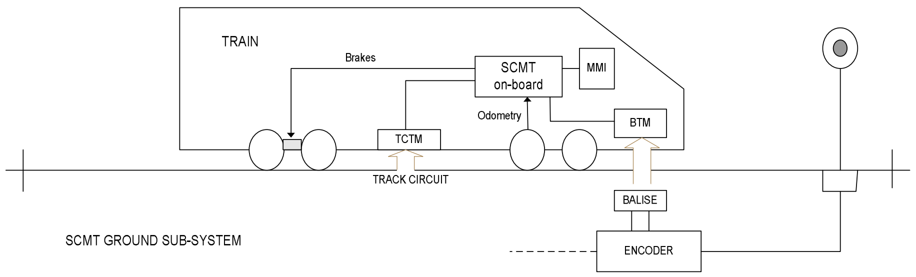


Fig. 1. SCMT Architecture Diagram

The whole SCMT system had to be tested in order to be validated against safety related requirements in both nominal and degraded operating conditions. In particular, to reduce testing complexity, it has been chosen to test separately the ground sub-system by verifying the correctness of both installation and data transmitted by balises. Therefore, SCMT system functional testing only regarded the on-board sub-system. However, as aforementioned, the main issue was that we could not completely rely on system requirements specification as it often did not extensively cover degraded conditions which constituted critical scenarios for system safety.

In the following, we will refer to the on-board sub-system as "SCMT on-board". In order to accurately select target system's input-output ports, we represent them

[2] Data transmitted by track-circuits is often referred to as "codes".

through a class diagram describing structural relationships involved in system architecture (see **Fig. 2**).

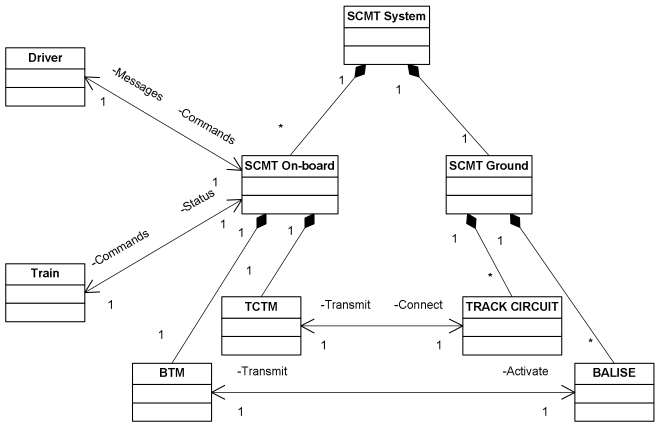
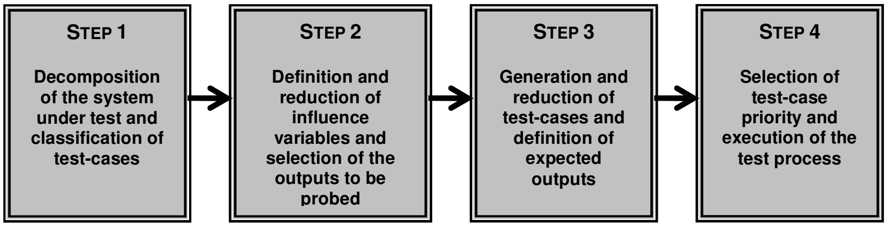


Fig. 2. Class diagram for SCMT on-board

Testing the on-board system for an exhaustive set of input sequences would be unfeasible for the exponential growth of test-cases. Such a problem, known in the literature as "test-case explosion", depends on the number of input variables and on the set of possible values for each variable. Next section will show how to contain test-case explosion applying proper reduction rules.

3 Description of the Testing Approach

A modular approach is commonly used when dealing with complex system development activities. In our approach there is a transposition of such a compositional methodology to functional testing issues: the decomposition of the system in distinct logic modules is made "a posteriori" and without knowing real system implementation. Logic modules have been separated whenever they could be proven to be independent and/or to interact with each other in a well defined way, according to system specification. Such a decomposition was then validated by analyzing structural dependencies within software modules and verifying that it respected the "a priori" assumptions. Function call-graphs based techniques have been adopted, together with traditional structural tests, to validate system decomposition [12, 13]. After this step, each module (or macro-function) had to be fed with extensive input sequences in order to check its output behaviour. Outputs had to be accessed by acting on system hardware by means of proper diagnostic instruments (hardware and software tools). It was important to ensure that such instruments were the less intrusive as possible, in order not to influence in any way system behaviour. This was achieved equipping the system under test with built-in hardware diagnostic subsystems and standard output communication facilities, in order to interface with software diagnostics. The followed approach is briefly summarized in **Fig. 3**.



STEP 1
Decomposition of the system under test and classification of test-cases
STEP 2
Definition and reduction of influence variables and selection of the outputs to be probed
STEP 3
Generation and reduction of test-cases and definition of expected outputs
STEP 4
Selection of test-case priority and execution of the test process

Fig. 3. Operational steps of the testing process

Step 1. The decomposition of the system into macro-functions (i.e. logic blocks) to be separately tested followed two directions (see **Fig. 4**):

- Horizontal decomposition, which was based on input-output relations;
- Vertical decomposition, by superposition of progressive complexity levels.

In the horizontal decomposition it was supposed that there was a well defined data path from inputs to outputs, i.e. a generic block was influenced only by external inputs and output of the previous block, but not by the above, below and following blocks. Vertical decomposition consisted in considering the system as the superposition of different working levels, from the simplest to the most complex, in a bottom-up way. The upper levels introduced new functionalities relying on a new set of external influence variables. In such a way, the system was divided in distinct logic blocks, for instance Braking Curve Elaboration (horizontal level) for Complete SCMT (vertical level). Each logic block was influenced by a well defined set of inputs and reacted with outputs that could be either accessible or not at the system's output interfaces. In case the output was not visible, internal probes had to be used to access the part of the system state we were interested in.

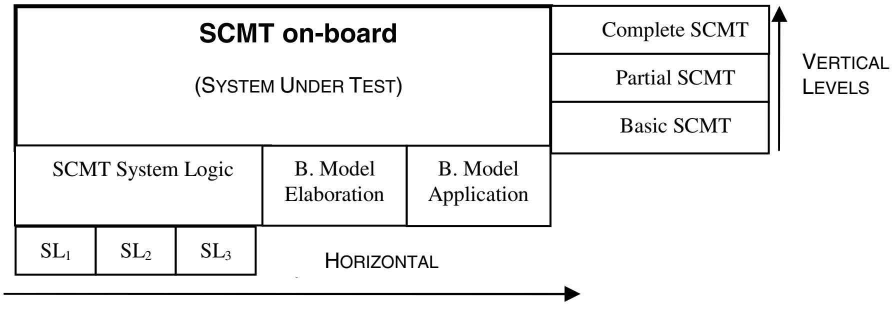


Fig. 4. SCMT decomposition into logic blocks

Therefore, a test-case classification according to the introduced logic blocks became possible. In other words, each block was associated with its own test suite. In particular, the first level horizontal decomposition of SCMT was constituted by the following blocks (see **Fig. 5**):

1. SCMT System Logic, which had the aim to apply the correct "work plane";
2. Braking Model Elaboration, aimed at building the proper protection curves;
3. Braking Model Application, which had to control the braking distance.

Even by not explaining the meaning of the variables exchanged between blocks (shown in **Fig. 5**), for the sake of simplicity, we would like to highlight that internal variables had to be assigned values, for testing purposes, acting on the corresponding external input variables. In order to perform this operation, we needed to know the functional behaviour of the previous block (obtainable by the specification) and to test it previously to ensure its correct implementation. For instance, having already tested the first horizontal block, it was straightforward to obtain the corresponding external inputs by means of a backward analysis from "internal" outputs (i.e. D_0, V_0, etc.) to external inputs (i.e. the real influence variables). This process could be iterated for every block.

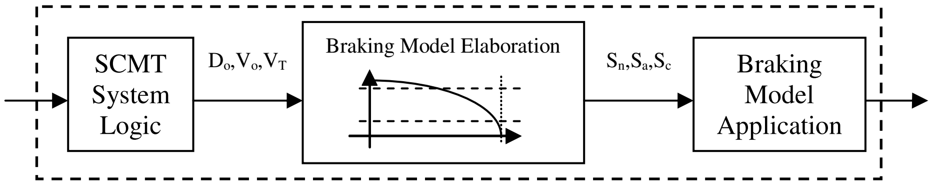


Fig. 5. First-level horizontal decomposition of SCMT in detail

SCMT vertical working levels are related to the completeness of the information received from the ground sub-system. At the Basic SCMT level, on-board sub-system only reads codes from track-circuits. At the Partial SCMT level, only information from static balises are added to the codes. Finally, in Complete SCMT level all ground transmission devices are used to collect data. It is important to underline that while real SCMT working levels contain incremental functionalities (e.g Partial SCMT comprises Basic SCMT), vertical levels shown in **Fig. 4** feature differential functionalities (e.g. Partial SCMT contains all the functions not already contained in the lower level). Such a distinction allowed us to reduce test redundancies, by verifying only the new functionalities at each working level, in a bottom-up way.

Finally, we were able to divide the target system into 15 blocks, with an influence variable average reduction factor of 2 for each block (further augmentable with other techniques). To estimate the achieved reduction factor, we considered a system characterized by the following parameters: *N*, total number of input variables; *m*, average number of possible values for each input variable; *s*, number of logic blocks in which the system has been divided; *r*, average reduction factor in the number of input variable for each logic block. For such system the reduction factor *R* in the total number of test-cases (and thus in the time required for their execution) is easily obtainable as follows:

$$R = \frac{\text{\# of tests before decomposition}}{\text{\# of tests after decomposition}} \cong \frac{m^N}{s \cdot m^{N/r}} = \frac{1}{s} m^{N \cdot (1-1/r)}$$

This expression proves that, as $r > 1$, the overall reduction factor grows with N in an exponential way. Thus, the effectiveness of the presented technique is particularly high when dealing with large systems, featuring a big amount of influence variables. For SCMT we achieved an overall reduction factor, at this stage, of more than 100.

Step 2. An influence variable is a variable which is able to influence the behaviour of the system under test. SCMT influence variables could be divided into the following two main groups:

- Scenario variables, which represented operating conditions (e.g. track circuit length and transmitted codes or code sequences, balises messages, etc);
- Test-Case variables, which represented specific inputs for a given scenario (e.g. train speed, key pressures, etc.).

At a first step, we identified all the possible influence-variables. Then, in order to reduce the number of influence variables for each logic block, we developed a simple procedure of variable-selection, with a step-by-step independence checking: if the value of a certain variable was directly deductible from the others, then it was excluded from the set of influence variables because it would have led to define further test-cases which, however, would have revealed equivalent to at least one of the already developed. For instance, for the proper working of the "SCMT System Logic" blocks, it was necessary to define a set of variables needed to express at least the following information: the completeness of the ground equipment (only track-circuits, track-circuits and static balises or track-circuits and dynamic balises), the type of installed balises and the consistency of the information contained in balises. On the basis of the requirements contained in system specification, we found out that the variable expressing that data consistency of balises was always dependant on the first two variables. Thus, such a variable was excluded from the set of influence variables, being redundant. Another simple example of a redundant variable consists in the one expressing the "Train Stopped" condition. Such a variable was always used in combination with "Train Speed", and its value was dependant on "Train Speed" value, because the train was considered stopped if and only if train speed was less than 2 Km/h.

Output variables classification was performed starting from the logic block(s) they influenced. In particular, to access "hidden outputs", that is outputs that were not normally accessible from the interacting entities (see **Fig. 2**), we needed to know the system physical structure. For instance, let us refer to the first level horizontal decomposition shown in **Fig. 5**. In such a case, the stimulating variables were D_o, V_o, V_T, while the output variables to be probed were S_n, S_a, S_c. The former variables were assigned values by properly acting on external accessible inputs, while the latter could be read from the log-files generated by the diagnostic software managing hardware probes. The diagnostic environment used for SCMT is depicted in **Fig. 6**.

Step 3. A tree-based test-case generation technique was applied to every logic block. At each level, only one influence variable, among the ones not already instantiated, was assigned all the significant values of its variation range, according to the reduction criteria that will be described later in this section. Influence variable instantiation could be divided into two macro-levels: scenario variables and test-case variables. In fact, test-cases were introduced only when all the significant operating scenarios had been defined. Such a process is represented in **Fig. 7**. The combination of the instances of the variables was performed automatically by a tool meant to apply a set of pre-determined reduction rules (described later on in this section), for an "a priori" pruning of pleonastic tree branches. With such an approach, tree leaves represented the test-cases which had to be actually executed.

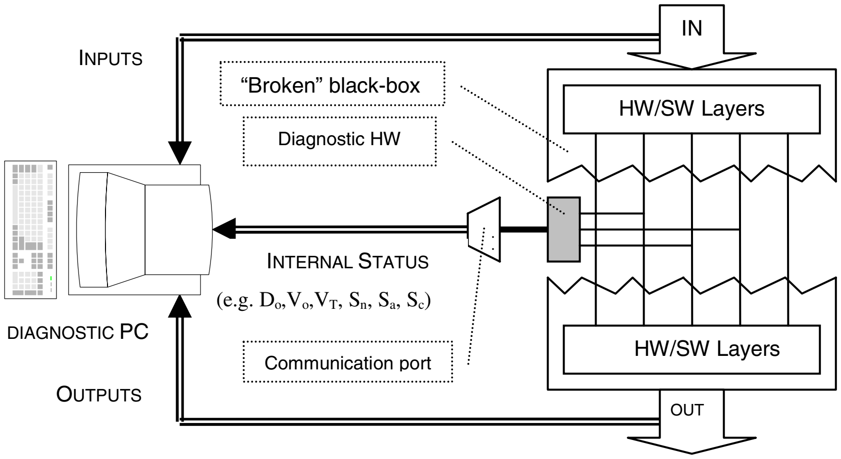


Fig. 6. Diagnostic environment

The main reduction criteria adopted have been:

- Incompatible combination of scenario or test-case variables;
- Not realistic operating conditions (for scenario variables);
- Equivalence class based reduction (considering scenario parameters or input variation ranges);
- Context specific reductions (i.e. context-specific dependencies, code-based static independence checking, mapping on test-cases already defined for a different logic block, etc.).

For scenario variables, the conditions to assess "incompatibility" were usually based on constraints coming from the physical environment, while "realistic operating conditions" refer to the railway national or international norms prescribing a set of requirements that must be necessarily respected [11]. Some of these norms are about track circuit length, balise positioning, signalling rules, and so on.

For test-case variables, context specific dependencies were very frequent and could be found when the assignment of some particular values or ranges of values to a specific set of variables implied a fixed value assignment for another distinct set of variables. Also code-based independence checking was used in order to avoid repetition of simple tests (e.g. key pressure failures) in different SCMT operating modes, when it could be proven that the called managing procedure was the same. Finally, as test-cases stimulated different logic-blocks, often only one execution was performed for multiple defined tests (what increased was the output checking time for the same test).

The most complex and efficient technique was based, both for scenario and test-case variables, on equivalence class based reductions. An equivalence class represents a set of valid or invalid states for a condition on input variables. The domain of input data is partitioned in equivalence classes such that if the output is correct for a test-case corresponding to an input class, then it can be reasonably deducted that it is correct for any test-case of that class. By tree-generating the combinations of influence variables and reducing them with an equivalence class based approach, we implemented what is called an extended SECT coverage criterion. SECT is the acronym of "Strong Equivalence Class Testing", which represents the verification of system behaviour against all kinds of class interactions (it can be extended with robustness checking by also considering non valid input classes). In our case, SECT was feasible because each logic block had a quite small number of influential variables, each one assuming a small number of classes of values.

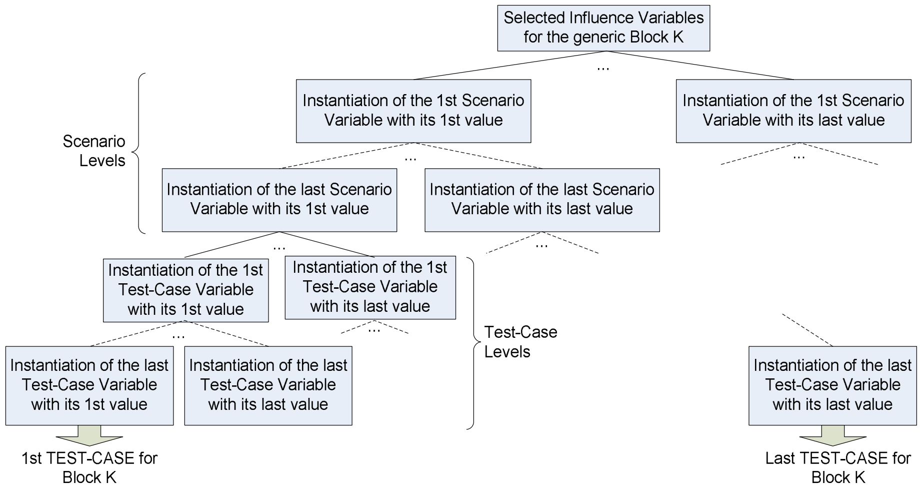

Selected Influence Variables for the generic Block K
Instantiation of the 1st Scenario Variable with its 1st value
Instantiation of the 1st Scenario Variable with its last value
Scenario Levels
Instantiation of the last Scenario Variable with its 1st value
Instantiation of the last Scenario Variable with its last value
Instantiation of the 1st Test-Case Variable with its 1st value
Instantiation of the 1st Test-Case Variable with its last value
Test-Case Levels
Instantiation of the last Test-Case Variable with its 1st value
Instantiation of the last Test-Case Variable with its last value
Instantiation of the last Test-Case Variable with its last value
1st TEST-CASE for Block K
Last TEST-CASE for Block K

Fig. 7. Tree-based test-case generation

Generally speaking, when we had to select input values for influence variables we chose at least one test-case for each of the following classes: internal values, high-boundary values, low-boundary values, near high-boundary values, near low-boundary values, over high-boundary values, below low-boundary values, special values (only for discrete variables). The last three classes were very important to test robustness (and thus to ensure system-level safety). All in all, a non Boolean variable assumed, in the final set of test-cases, at least three different values, belonging to the first three categories. The followed general approach included "Boundary Analysis", "Robustness Testing" and "Worst Case Testing" (WCT) techniques[3]. In some cases, according to system specifications, we merged the three techniques by adopting nearness, robustness and worst-case conditions. For instance, train speed ranges have been first partitioned into sub-sets, according to the specified speed limits, that is: V<2 Km/h, 2Km/h<V<15Km/h, 15Km/h<V<30Km/h, etc. The obtained sub-sets represented all the significant train speed ranges used in the entire system specification. However, for all tests in which we were testing a function which only required the train to be under a maximum speed limit (V_{MAX}), train speed values were chosen as follows: $V=1/2\ V_{MAX}$; $V=V_{MAX} - \delta V$; $V=V_{MAX} + \delta V$ (where δV is a small positive speed value, for instance 1Km/h). The so defined speed values were combined with all the remaining influence variable values, even in the worst-case conditions.

The next stage consisted in determining expected outputs for each test-case. Examples of measured output variables are: leds (on/off), icons, text messages, brake intervention, train position, etc. System behaviour was modelled in terms of significantly varying outputs and their expected values. The correct behaviour was directly obtainable from system specifications, but, in some cases, it had to be derived by means of a parallel independent model. Comparison of the results was then made manually. For instance, we used a parallel independent model for the braking curve prediction based on a human comparator. When the described approach was able to highlight incompleteness or incoherence in system specifications, it was necessary that the responsible for system specifications assessed the correct system behaviour corresponding to the identified input conditions. Finally, as an operational tool, we made use of a spreadsheet in order to represent Test Case Specification in terms of: operating scenario, represented by instantiated scenario variables (e.g. TC code sequence: CODE1→CODE2); input sequences, represented by test-case variable values and their associated time-line (e.g. key X pressed within time T); expected outputs and optional measurement instructions (e.g. icon Y appearing on the MMI display and recorded as variable V_y in log-file L).

Step 4. A relevant amount of tests had to be executed on the SCMT system, so it was useful to identify priority levels for test classes. The main criterion was the safety criticality of functions/blocks, identified by the hazard-analysis processes [9], used to validate system specification against the most critical causes of dangerous failures. For instance, correctness of system behaviour was first tested against the so called "restrictive input sequences", that is track-circuit code sequences that should activate one or more train protection mechanisms. Moreover, in testing the system for any new software version, we adopted a "width-first" coverage strategy, that consisted in executing the most significant test-cases for each block and category, in order to quickly discover and notify macroscopic non conformities or dangerous behaviours.

Test-cases have been executed in a simulation environment made up by: the system prototype under test; hardware devices simulating external interactions (i.e. system inputs); software tools aimed at simulating the operating environment (i.e. the scenario) and allowing the automation of the test process through batch-files. To

[3] Such techniques are based on empirical studies. For instance, it has been noted that most errors generate in correspondence of extreme values of input variables.

speed up test process, preparation, execution and output verification activities have been pipelined, in order to allow more test-engineers work in parallel. Comparison of the output log-files with the expected outputs had to proceed manually in the first phase of test execution, because Pass/Fail criteria were often based on time/speed ranges, very difficult to validate in a complete automatic environment. In all cases a faulty behaviour has been observed, a proper notification (i.e. System Problem Report) was sent to the development division. Such a notification could regard, as aforementioned, implementation errors as well as specification errors. The testing environment used for SCMT has been depicted in **Fig. 8**.

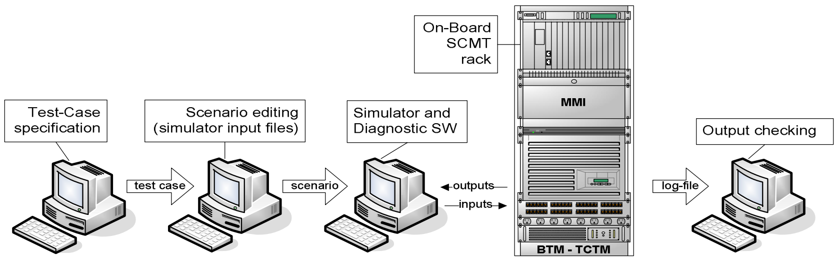


Fig. 8. Test execution pipeline

In order to verify the correct implementation of needed modifications and to ensure the absence of regression errors, the whole set of tests had to be repeated at any new software version. This is the most simple and safe non-regression testing technique, known in literature as "Retest-All" [14]. A regression testing technique consists in selecting a sub-set of the entire set of test-cases to be repeated on modified software versions. We chose not to implement other safe techniques for the following reasons:

- Any safe technique requires a relevant amount of time to identify the set of test-cases to be repeated, while the test-suite reduction can be often very small, if any (see [14]). So there is a risk of over-estimating cost-effectiveness of safe techniques;
- As test-execution was automated, test-suite reduction was not a fundamental concern.

Of course, it did remain the problem of checking the correctness of output log-files. We overcame the problem by implementing a so called Gold-Run technique [15]. A Gold-Run (or Gold-Standard) log-file contains the results of a test which showed no unconformities. While the comparison with expected outputs, as aforementioned, had to proceed manually, the verification of log-files corresponding to repeated test-cases could be automated by means of a software comparator tool. The building of the tool was complicated by non deterministic outputs (such as time delays or non rigid output sequences). Comparison on stochastic values was based on confidence ranges calculated by estimating the variance of the results obtained in more test runs (typically three runs were enough). This implied some more test runs and manual verifications, but ensured a better level of automation, with a reduced error rate. Of course, a manual control had to be pursued in case of failures, in order to ensure the absence of positive faults, that is faults caused by the automatic comparator tool and not by systematic errors in the target system.

4 Conclusions and Future Work

The validation of the SCMT system implementation has been a multi-month activity involving several highly specialised resources. The approach described in this paper allowed to define and execute more than 2500 test-cases, covering all the functionalities of the system in normal as well as in degraded states of operation. With this activity a number of errors were revealed, and this contributed to improve system specifications in terms of both correctness and completeness. While specification completeness verification was among the objectives of the methodology, correctness checking constituted an important collateral activity. By describing our industrial experience, we showed how a hybrid functional testing methodology can be adopted in order to achieve different testing objectives. In fact, a combination of different techniques have been used during the entire SCMT test process. In particular, the developed grey-box approach has proven to be able to improve test coverage while reducing the number of required test-cases. Test process has been accompanied by an extensive documentation activity (test plan, test design specification, test case specification, etc.) that revealed the importance of test organization in speeding up execution and in allowing easy test reproducibility. The verification of coverage by code instrumentation is still in progress, but so far it showed a coverage (using a Decision to Decision Path technique [13]) of nearly 90%, which did highlight the effectiveness of the followed testing approach. The uncovered parts of software are going to be tested by adding the necessary system-level tests. They gave us an important feedback on potential errors in classifying influence variables or in defining and applying some of the reduction rules. We are continuously developing techniques and tools to further automate the testing process, above all by optimizing the output checker. The presented testing methodology, with the necessary customizations, is now being applied to validate the new ERTMS/ETCS [10] compliant systems developed by ASF.

References

1. W. S. Heath: Real-Time Software Techniques. Van Nostrand Reinhold, New York (1991)
2. CENELEC: EN 50126 Railways Applications – The specification and demonstration of Reliability, Maintainability and Safety (RAMS)
3. G. J. Myers: The Art of Software Testing. Wiley, New York (1979)
4. J. Wegener, K. Grimm, M. Grochtmann: Systematic Testing of Real-Time Systems. Conference Papers of EuroSTAR '96, Amsterdam (1996)
5. B. Jeng, E.J. Weyuker: Some Observations on Partition Testing. In Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification, Key West (1989)
6. T. Ostrand, M. Balcer: The Category-Partition Method for Specifying and Generating Functional Tests. Communications of the ACM, 31 (6), (1988) 676-686

7. M. Grochtmann, K. Grimm: Classification-Trees for Partition Testing. Journal of Software Testing, Verification and Reliability, Vol. 3, No.2, (1993) 63-82
8. K. Grimm: Systematic Testing of Software-Based Systems. In Proceedings of the 2nd Annual ENCRESS Conference, Paris (1996)
9. P. di Tommaso, R. Esposito, P. Marmo, A. Orazzo: Hazard Analysis of Complex Distributed Railway Systems. In Proceedings of 22nd International Symposium on Reliable Distributed Systems, Florence (2003) 283-292
10. UNISIG ERTMS/ETCS – Class1 Issue 2.2.2 Subset 026-1
11. Ministero dei Trasporti – Ferrovie dello Stato – Direzione Generale: Norme per l'Ubicazione e l'Aspetto dei Segnali
12. I. Sommervill: Software Engineering, 6th Edition. Addison Wesley (2000)
13. Telelogic Tau Logicscope v5.1: Basic Concept. (2001)
14. T. L. Graves, M. J. Harrold, J. M. Kim, A. Porter, G. Rothermel: An Empirical Study of Regression Test Selection Techniques. In Proceedings of the 20th International Conference on Software Engineering (1998) 188-197
15. E. Dustin, J. Rashka, J. Paul: Automated Software Testing, Addison Wesley (1999)
16. Bred Pettichord: Success with Test Automation. In Quality Week, San Francisco (2001)

Deterministic Test Vector Compression / Decompression Using an Embedded Processor[†]

Maciej Bellos and Dimitris Nikolos

Technology and Computer Architecture Laboratory,
Dept. of Computer Engineering & Informatics, University of Patras,
26500 Patras, Greece
Research Academic Computer Technology Institute, 61 Riga Feraiou Str.,
26221 Patras, Greece
bellos@ceid.upatras.gr, nikolosd@cti.gr

Abstract. Test data compression and on-chip decompression using an embedded processor has already been proposed for test data volume and test time reduction as well as for use of slower testers without decreasing test quality. On the other hand, scan cell reordering methods have been proposed to overcome the problem of high average power dissipation during scan based external testing. In this paper we propose a scan cell ordering based test vector compression method, which reduces test data volume up to 87.3%. The decompression of the test data is based on the use of an embedded processor.

1 Introduction

Shrinking process technologies and increasing design sizes have led to highly complex, billion-transistor integrated circuits. The latest System-on-Chip, SoC, designs integrate multiple ICs (microprocessors, memories, DSPs and I/O controllers) on a single piece of silicon. Each one of them must be exercised by a large number of precomputed test patterns. One of the increasingly difficult challenges in testing SoCs is dealing with the large amount of test data that must be transferred between the tester and the chip [1]. The large amount of test data is not only exceeding the memory and I/O channel capacity of commercial automatic test equipment (ATE) but is also leading to excessively high testing times.

Several techniques have been proposed for test data volume reduction. Test data compression is achieved by using techniques such as statistical coding, run-length coding, Golomb coding, Huffman coding and frequency directed coding [2-9]. The test data decompression is performed on chip using a decoder. Test data compression is also achieved by statistical coding combined with LFSR reseeding [10], by decompression network based on linear equations [11], by SISR combined with spreading logic [12] and by ring generators combined with phase shifters [13]. TestKompress®is a commercial tool by Mentor Graphics Corp., which implements the work presented

[†] This research was financially supported by the Public Benefit Foundation "Alexander S. Onassis" via its scholarships program and by the EPEAEK II "Pythagoras" program.

in [13]. The methods proposed in [10-13] take advantage of the presence of don't care bits in order to achieve significant compression ratios. Another approach of reducing the amount of test data that needs to be transferred to the SoC is by using the processing power already present in most of the contemporary systems [14-19]. Some techniques have been proposed for testing embedded memories [14, 15]. In [16] an embedded processor is used to simulate multiple-polynomial multiple-seed LFSRs in order to detect as many faults as possible before switching to deterministic vector application. In [17] an accumulator based BIST is proposed that achieves less memory requirements than that of [16]. The basic approach of how an embedded processor is used for testing core-based SoCs is presented in [18]. An external tester is used to download test data to the memory of the SoC and then the embedded processor acts as a controller assuming the tasks of coordinating the test data download, accessing the appropriate core and applying the test data. The authors of [19] present a specific compression/decompression algorithm, which gives good results, and show how an embedded processor is used to decompress and apply the test data on the cores of the SoC. An external tester loads a program along with compressed test data into the processor's on chip memory. The processor executes the program, which decompresses the test data and applies it to scan chains in the other components of the SoC to test them. The decompression process requires very few processor instructions and thus can be done very quickly. This approach reduces both the amount of data that must be stored on the tester and reduces the test time. Moreover, even if a slow tester can be used to download the test data to the memory, the test data can be shifted in at-speed during the testing of the cores.

Test data compressed using one of the techniques presented in [2-13] could also be decompressed by an embedded processor. However, the delay imposed by the execution of the required processor instructions would diminish the advantage of reduced test application time that these techniques offer.

In this work we propose a more efficient algorithm than the one proposed in [19] for compressing test data. The compression algorithm is based on test vector ordering, rearranging of the columns of the vector table and don't care bit assignment. The decompression process can be performed easily and fast by an embedded processor. In Section 2 we briefly describe the scheme that is used for data representation and how the embedded processor is used for decompression of the test data. Section 3 presents the proposed algorithm while Section 4 provides experimental results. Finally Section 5 concludes.

2 Previous Work

In the method proposed in [19] the embedded processor generates the next vector from the previous one using test data downloaded from ATE. This is based on the differences of the current test vector with respect to the previous one. If, however, the differences of the two vectors are expressed in the form of the bit place where the two vectors differ then this will lead to an excessive amount of information, which will surpass the amount of information saved by the common bits. In order to overcome this problem the test vector is divided in blocks of fixed length *w*, each of which refers to the same bit positions in each test vector. Therefore, one only needs to know which blocks of the next vector are different from the corresponding blocks of the previous vector and the next vector is produced by replacing these blocks, as shown in Figure 1. The shaded blocks indicate the blocks that were replaced in vector *n* in order to produce vector *n+1*.

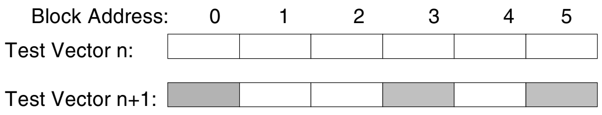


Fig. 1. Test vector generation principle

In order to achieve large savings we need to have the least possible number of different blocks between pairs of successive test vectors. In [19] this is achieved by ordering the test vectors in such a way that successive test vectors differ in a rather low number of blocks. Since the faults in the circuit are structurally related, there are test vectors that have large common parts and thus the ordering of the test vectors can significantly reduce the number of different blocks needed for the generation of the test set. The ordering procedure used, was based on using a nearest neighbor algorithm and the criterion for choosing the next vector was the number of blocks the two vectors differ.

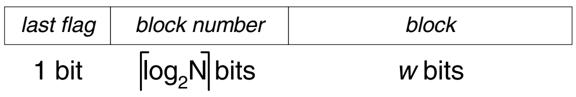


Fig. 2. Replacement word structure

After the ordering of the vectors the useful information of the test set is comprised of the different blocks each two successive vectors exhibit. These blocks will be used by the embedded processor in order to reproduce the given test vectors. Each such block is accompanied by two fields. The first, denoted as *last flag*, is a single bit field, which indicates whether the block is the last block of the test vector. The second, denoted as *block number*, holds the position in the test vector where the block must be placed. If a vector is divided in *N* blocks then the latter field requires $\lceil \log_2 N \rceil$ bits, where $\lceil x \rceil$ denotes the upper integer of x. The block together with the two extra fields is denoted as a "replacement word" (Figure 2) and the total amount of data required to encode the initial test set is equal to the number of replacement words multiplied by the size of the replacement word. As an example, consider that the test vector width is equal to 50 bits and that we want to divide it in blocks of 10 bits each. Therefore the vector is divided in 5 blocks which in turn dictates that the *block number* field is equal to $\lceil \log_2 5 \rceil$ = 3 bits. Considering also that the *last flag* field is equal to 1 bit we get that the replacement word is equal to 14 bits.

The total number of the replacement words will probably be quite large and thus it is not possible to store it on chip in a ROM. However the embedded RAM can be used to store some of the replacement words, which will be processed by the embedded processor at a latter stage. In order to provide the appropriate data to the processor an external tester is used to download replacement words to the on-chip memory. These words are then processed by the embedded processor during the procedure of test generation. Care needs to be taken so as to avoid memory overflow, that is, not to download test data at a rate that the processor cannot handle.

3 Proposed Scheme

The scheme for compression/decompression of test data proposed in this work is based on the transformation of the test data to a desirable form via a number of steps and the use of an embedded processor as in [19]. The goal is to reduce the amount of test data transferred to the embedded processor through the use of ATE. The test data is organized as a vector table T containing *n* vectors of *w* bits each. The transformation of vector table T aims to produce a new version of T that can be efficiently encoded by the use of the information representation proposed in [19]. The algorithm that achieves this goal is comprised from the following three steps:

1. Test vector ordering, which is equivalent to the Traveling Salesman Problem (TSP), known to be NP-complete.
2. Rearrangement of the columns of the table, which represent the bit values each circuit input receives. These columns will be henceforth denoted as bit columns.
3. Proper assignment of the don't care bits.

3.1 Test Vector Ordering

The rearrangement of the bit columns, which follows the test vector ordering step in the proposed scheme, does not permit the ordering of the vectors using the same criterion as in [19], since the number of different blocks between two vectors may not be equal to the number of different blocks after the rearrangement of the bit columns. The criterion used instead is the number of different bits a vector pair has, i.e. the Hamming distance of the test vector pair. The computation of the Hamming distance of a test vector pair takes into account the existence of don't care bits. In this case the don't care bit is assumed to have the same value as the corresponding bit of the other vector. For example, the Hamming distance of vectors w = 1x011xx1 and z = x0101100 is equal to 3. Assignment of specific values to the encountered don't care bits is not applied at this point, since we don't know where the vectors containing the don't care bits will be placed after the ordering procedure. For example if a don't care bit is assigned value 1 and after the vector ordering process this bit has two neighboring bits of value 0 then two changes of value are encountered in that bit column which may cause a succession of 3 different blocks.

Using the computed Hamming distances we construct an undirected weighted transition graph TG(V, E), where each vertex v∈V represents a test vector and each edge e∈E has a weight equal to the Hamming distance of the two test vectors it connects in the graph. The Greedy algorithm uses graph TG as input and starts from an empty set of edges denoted as *Order*. At first, the edge with the lowest weight is added to *Order*. Then, the edge (*i*, *j*) with the lowest weight is chosen from the remaining edges and it is checked if it fulfills two criteria. The first one is that the addition of edge (*i*, *j*) does not make neither *i* nor *j* obtain a degree larger than 2, that is the number of edges starting from each of the two nodes must be strictly less than or equal to 2. The second criterion is that the addition of edge (*i*, *j*) does not create a cycle in *Order*. The two criteria assure that we will eventually construct a path starting from a vertex of TG and ending at another vertex of TG while visiting all vertices. The edge selection ends if the edges of *Order* cover all vertices of graph TG or equivalently if *Order* contains *n*-1 edges, with *n* being the number of test vectors. Finally using set *Order* the algorithm creates a sequence of vectors that still contain don't care bits, since no don't care bit assignment is performed.

For example, consider the graph of Figure 3. Edge (a,c) has the lowest weight and thus is the first edge added to *Order*. Then edges (b, c) and (b, e) are added. Edge (c, d) cannot be added since it violates the first criterion. The next edge examined is (b, d) which cannot be added since it violates the second criterion. The same applies for edge (a, e). Edge (e, d) is considered next and since it fulfills both criteria it is added to *Order*. With this addition we cover all vertices of TG and thus the final state of *Order* is {(a, c), (b, c), (b, e), (d, e)}. Using *Order* and starting from the first inserted edge, we create the following sequence of vectors: {a, c, b, e, d}

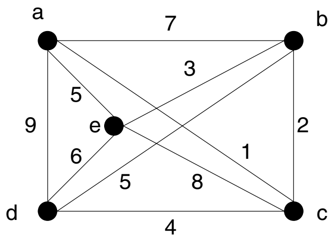


Fig. 3. Graph used in the Greedy ordering example

The significance of postponing the assignment of specific values to the don't care bits will be outlined in the third step of the algorithm. Obviously since we haven't rearranged the bit columns we have no knowledge of where the don't care bits will be placed and therefore assigning specific values is not appropriate. Once the bit columns are located in the desired positions we can cleverly assign specific values that will lead to the reduction of the blocks needed to reproduce the given test set.

Assume that we have the test vector table *T* shown in Figure 4.a. Consider also that each test vector is divided into two blocks of 4 bits each, as indicated by the vertical dotted line. With the present vector order in mind, the number of blocks that need to be downloaded from the tester to the SoC's memory so as to reproduce the test set is equal to 10. According to the test vector ordering procedure we construct a weighted transition graph by computing the Hamming distances of all possible pairs of test vectors. We then use the Greedy algorithm to order the test vectors, which results in table *T'* of Figure 4.b. If we proceeded with don't care bit assignment at this point we would require 9 blocks for the reproduction of the test set of Figure 4.b. However, as we have already reported, the don't care bit assignment to specific values is postponed until the completion of the bit column rearrangement step.

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
v_1	1	x	1	1	0	0	1	0
v_2	0	0	1	0	1	1	0	1
v_3	1	x	1	0	0	x	1	0
v_4	1	0	0	1	0	x	0	1
v_5	1	0	0	0	0	x	0	0

a) Initial Table *T*

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
v_1	1	x	1	1	0	0	1	0
v_3	1	x	1	0	0	x	1	0
v_5	1	0	0	0	0	x	0	0
v_4	1	0	0	1	0	x	0	1
v_2	0	0	1	0	1	1	0	1

b) Table *T′* after vector ordering

Fig. 4. Test vector ordering step

3.2 Rearrangement of the Bit Columns

Having ordered the test vectors we can now proceed with the step that rearranges the bit columns of the new vector table T′. The goal is to reduce the number of different blocks between two successive test vectors. Some bit columns of T′ will have a small number of changes, i.e. transitions, from 0 to 1 and 1 to 0, while other bit columns will have a lot of changes from 0 to 1 and 1 to 0. Let us consider one of the latter columns, denoted as c_p. Since we have divided each test vector in blocks of bits we consider the blocks of the vectors that bit column c_p participates in. Obviously if this bit column has different values in test vectors *i* and *i*+1 then the two blocks this bit column participates in are different. Therefore if the column exhibits *l* changes in its values it will yield *l* different blocks in the *n* test vectors of T′. If another bit column c_q of the blocks has *m* changes in its values then the two columns will be responsible for at most *l*+*m* different blocks, since there may be changes of values in some common places for each of the two columns. With the above observation in mind we can group the columns in blocks of fixed width according to the number of value changes each column exhibits. This increases the possibility of forming groups of corresponding blocks, that is blocks that belong at the same position in different test vectors, where there are a lot of different blocks and a lot of groups where the majority of blocks are the same and in consecutive test vectors, thus leading to an overall decrease in the number of different blocks required for the reproduction of the test set. Although the rearrangement of the bit columns is based on the number of changes in each column and ignores where the changes actually occur, we will see in the experimental results that it works quite well.

The procedure that rearranges the columns is comprised from two parts. The first is to compute the number of changes of value between successive bits in each column and assign it as a weight to the corresponding column. Since don't care bits are present, the computation of the weight of each column needs to consider them also. In the case where don't care bits exist between different values, that is 0x…x1 or 1x…x0, we add one to the weight of the column and we leave the bits unassigned. In all other cases the don't care bits are assumed to have the same value with their neighboring defined bits and the weight of the column remains unchanged. The second part sorts the columns of the test set in descending order according to their respective weights.

During the decompression phase the bits of the test vectors must be rearranged so that they can be loaded into the scan chains in the correct order. The embedded processor could also undertake the task of placing the bits in their correct positions before shifting in the test vectors to the scan chains, however this would result in large decompression time, since the use of time consuming bit manipulation functions combined with lookup up tables would be required. A solution to this problem is to suitably order the scan cells of each core of the SoC, so that the downloaded vectors are properly loaded in the scan cells. Scan cell ordering has been already proposed for average power dissipation and energy consumption minimization during the shift-in/shift-out phase of scan based testing [20-23]. In this work we use scan cell ordering to target test data compression.

However, rearranging the scan cells may lead to area overhead and/or violation in timing constraints due to increase in routing. In order to exclude or alleviate these consequences, we examine also the following bit column rearrangement policies:

a) Only a percentage of the bit columns, namely the ones with the largest weight, is sorted. The rest bit columns keep their relative positions, that is, if column *p* appears before *q* in the original test set then it also appears before *q* in the rearranged test set. This policy leads to a small increase in routing, which in turn implies small area and delay overhead but does not guarantee avoiding timing constraint violation.
b) Some bit columns are required to remain at their original position while we can sort the rest bit columns in descending order. The sorted bit columns are placed in the positions that are not allocated for the bit columns that cannot be moved. By selecting the cells that belong to the critical paths so that they remain in their original position, we guarantee that the design meets the desired timing constraints.
c) Sorting of the columns is performed only within clusters of consecutive bit columns. The size of each cluster is equal to k**w*, where k = 2, 3, … *N*. Obviously some of the bit columns can stay at the same position within the cluster. This policy reduces the area and delay overhead imposed.

Using the new order of the bit columns we group them in blocks of the length *w*.

Applying the bit column rearrangement step on test set *T′* of Figure 4.b we get the table depicted in Figure 5. The bit columns were rearranged considering that all columns can be moved and that all columns are sorted in descending order. Thus the leftmost (rightmost) bit column exhibits the largest (smallest) number of transitions.

	c_4	c_3	c_1	c_5	c_6	c_7	c_8	c_2
v_1	1	1	1	0	0	1	0	x
v_3	0	1	1	0	x	1	0	x
v_5	0	0	1	0	x	0	0	0
v_4	1	0	1	0	x	0	1	0
v_2	0	1	0	1	1	0	1	0

Fig. 5. Bit column rearrangement

3.3 Proper Assignment of the Don't Care Bits

The above two steps have altered the arrangement of the test vectors and the arrangement of the bit columns in the test vector table. They were based on assumptions made about the values of the don't care bits without, however, assigning a specific value to each one of them. Upon completion of both the above steps we can now assign specific values to the don't care bits. The don't care bits are divided into two categories. The first category contains the don't care bits that reside in between defined bits of the same value or are located at the start or the finish of a column, while the second category contains all the remaining don't care bits. The bits of the first category are assigned the value of the bit that is either before the beginning or after the end of the sequence of the don't care bits. The don't care bits of the second category, that is, those that reside in between defined bits of different value, must be tackled carefully since a change from 1 to 0 or vice versa will occur somewhere in the sequence of the don't care bits. This change in values must be carefully placed so as not to change two identical neighboring blocks. Therefore we search for the first occurrence of two consecutive blocks that are already different in their defined bits. The don't care bits up to the first block are assigned to the value of the defined bit that resides before the beginning of the don't care bits sequence while the rest are assigned the value of the defined bit that resides after the end of the don't care bit sequence.

In order to illustrate the impact of the don't care bit assignment consider the test set table of Figure 5. The goal is to remove the don't care bits so as to obtain the least possible different blocks present. Column c_6 has 3 don't care bits at vectors v_3, v_5 and v_4 between bits of different value. An assignment of values as the one of Figure 6.a would yield a test set that would require 9 blocks to be downloaded. However, we can observe that vectors v_3 and v_5 already have different defined bits in the rightmost block and therefore their corresponding don't care bits can be assigned to different values. In this way the number of required blocks is reduced to 8 (Figure 6.b). Obviously the don't care bits of column c_2 are assigned to value 0.

	c_4	c_3	c_1	c_5	c_6	c_7	c_8	c_2
v_1	1	1	1	0	0	1	0	**0**
v_3	0	1	1	0	**1**	1	0	**0**
v_5	0	0	1	0	**1**	0	0	0
v_4	1	0	1	0	**1**	0	1	0
v_2	0	1	0	1	1	0	1	0

a) Improper bit assignment

	c_4	c_3	c_1	c_5	c_6	c_7	c_8	c_2
v_1	1	1	1	0	0	1	0	**0**
v_3	0	1	1	0	**0**	1	0	**0**
v_5	0	0	1	0	**1**	0	0	0
v_4	1	0	1	0	**1**	0	1	0
v_2	0	1	0	1	1	0	1	0

b) Proper bit assignment

Fig. 6. Test data transformation example

4 Experimental Results

Several experiments were performed for evaluating the proposed method using the largest ISCAS '89 benchmark circuits. The proposed scheme was used to compress the test sets of some of the ISCAS '89 benchmark circuits. We used the ATALANTA

ATPG tool [24] to generate test cubes that provide 100% fault coverage. Then a simple static compaction procedure was used in order to reduce the number of test cubes. The resulting test set T was then fed to the vector ordering, bit columns arrangement and don't care bit assignment procedures, which produced the number of replacement words needed for the compression of T. For all the test sets we have used a replacement word size of 32 and 64 bits, which in turn determined the size of the blocks each test vector was divided into.

Table 1 contains experimental results for replacement words of 32 and 64 bit width. Column 1 shows the circuit name while columns 2 – 4 show respectively the number of test vectors, the number of scan cells and the number of the initial test data bits. Columns 5 and 8 show the requirements in replacement words and columns 6 and 9 show the size in bits of the compressed test set for 32 and 64 bit replacement words respectively. Finally columns 7 and 10 show the attained compression ratio for 32 and 64 bit replacement words, which is calculated as:

$$(1 - \text{Compressed Bits/Initial Bits}) * 100\%$$

Table 1. Compression ratio attained with the proposed work for the ISCAS'89 circuits

				32 bit replacement words			64 bit replacement words		
Circuit	Vectors	Scan Cells	Initial Test Bits	Replacement Words	Compressed bits	Savings	Replacement Words	Compressed bits	Savings
s5378	131	214	28034	382	12224	56.4%	227	14528	48.2%
s9234	188	247	46436	653	20896	55.0%	387	24768	46.7%
s13207	263	700	184100	732	23424	87.3%	515	32960	82.1%
s15850	169	611	103259	880	28160	72.8%	544	34816	66.3%
s38417	184	1664	306176	3740	119680	60.9%	1999	127936	58.2%
s38584	165	1464	241560	2542	81344	66.3%	1387	88768	63.3%

We can observe that the proposed scheme achieves significant reduction in storage requirements, which for the case of s13207 reach up to 87.3% and 82.1% for 32 and 64 bit replacement words respectively. It is also notable that the increase in replacement word size and consequently in the block size reduces the compression ratio achieved. This is due to the fact that when the block size increases it becomes more difficult to find identical blocks in the test data.

Table 2. Comparison against compacted test set obtained from Mintest [25]

		32 bit replacement words		64 bit replacement words	
Circuit	Mintest bits	Bits	Reduction	Bits	Reduction
s5378	20758	12224	41.1%	14528	30.0%
s9234	25935	20896	19.4%	24768	4.5%
s13207	163100	23424	85.6%	32960	79.8%
s15850	57434	28160	51.9%	34816	39.4%
s38417	113152	119680	-5.8%	127936	-13.1%
s38584	161040	81344	49.5%	88768	44.9%

We have also examined how the proposed method fares against the smallest ATPG-compacted test set known for the circuits we have considered. We compare the test data volume attained with the proposed method against the compacted test sets that were obtained using the Mintest ATPG tool [25]. As it can be seen from Table 2 in all, but one, cases the proposed method achieves significant savings in test data volume requirements that can reach up to 85.6% and 79.8% when the replacement word size is equal to 32 and 64 bits respectively.

In order to evaluate the effectiveness of the proposed scheme we compared it against the scheme proposed in [19]. Experimental results are listed in Table 3. Columns 2 and 3 show the replacement word requirements for the scheme proposed in [19] and the proposed scheme respectively when 32 bit replacement words are used. As it can be seen the proposed scheme that combines vector ordering and bit columns rearrangement can significantly reduce the storage requirements when compared with the scheme proposed in [19] which only performs vector ordering. The reduction reaches up to 39.6% for the case of the s5378 benchmark circuit. For replacement word size equal to 64 bits the savings reach up to 42.2% for the case of the s38584 benchmark circuit. Therefore the proposed scheme alleviates to a certain degree the negative effect of the increase in replacement word size that was observed in [19].

Table 3. Test data volume comparison

	32 bit replacement words			64 bit replacement words		
Circuit	[19]	Proposed	Savings	[19]	Proposed	Savings
s5378	632	382	39.6%	374	227	39.3%
s9234	914	653	28.6%	575	387	32.7%
s13207	986	732	25.8%	711	515	27.6%
s15850	1163	880	24.3%	761	544	28.5%
s38417	5556	3740	32.7%	3252	1999	38.5%
s38584	3995	2542	36.4%	2398	1387	42.2%

Rearranging the scan cells may lead to area overhead and/or violation in timing constraints due to increase in routing. Therefore, we carried out experiments following the policies *a*, *b* and *c*, which where described in subsection 3.2. According to policy *a* only a percentage of the scan cells is rearranged with respect to the number of transitions, while according to policy *b* a percentage of the scan cells cannot move from the position it resides in the original scan chain arrangement. In the first case we rearrange the scan cells whose respective bit columns exhibit the largest number of transitions. Rearranging only a small portion of the scan cells will leave the largest part of the scan chain intact and will alleviate the problem of excessive routing. The second case considers the fact that moving certain cells from their original position may cause violation timing constraints and therefore it is best to leave them in tact.

Tables 4 and 5 show the replacement words required for policy *a* when 10%, 20% and 40% of the scan cells are rearranged for 32 bit and 64 bit replacement word size.

As it can be seen, even for the case where only 10% of the scan cells are rearranged the reduction in replacement word requirements can reach up to 28.6% for 32 bit replacement words and up to 31.7% for 64 bit replacement words when compared with the scheme proposed in [19]. Furthermore if we compare Tables 4 and 5 we can observe that the reduction achieved for a given percentage of rearranged scan cells is in the general case larger when the replacement word is increased to 64 bits and reaches up to 42.2% when all the scan cells are rearranged. As the replacement word size increases the proposed method becomes even more efficient than that proposed in [19].

Table 4. 32 bit replacement word requirements when arranging a percentage of the scan cells

		Proposed Scheme with 32 bit replacement words							
Circuit	[19]	10%	Savings	20%	Savings	40%	Savings	100%	Savings
s5378	632	474	25.0%	420	33.5%	405	35.9%	382	39.6%
s9234	914	750	17.9%	697	23.7%	686	25.0%	653	28.6%
s13207	986	821	16.7%	773	21.6%	743	24.7%	732	25.8%
s15850	1163	975	16.2%	922	20.7%	904	22.3%	880	24.3%
s38417	5556	3969	28.6%	3920	29.5%	3829	31.1%	3740	32.7%
s38584	3995	2862	28.4%	2672	33.1%	2617	34.5%	2542	36.4%

Table 5. 64 bit replacement word requirements when arranging a percentage of the scan cells

		Proposed Scheme with 64 bit replacement words							
Circuit	[19]	10%	Savings	20%	Savings	40%	Savings	100%	Savings
s5378	374	298	20.3%	248	33.7%	242	35.3%	227	39.3%
s9234	575	511	11.1%	461	19.8%	423	26.4%	387	32.7%
s13207	711	563	20.8%	547	23.1%	511	28.1%	515	27.6%
s15850	761	610	19.8%	582	23.5%	562	26.2%	544	28.5%
s38417	3252	2354	27.6%	2241	31.1%	2132	34.4%	1999	38.5%
s38584	2398	1639	31.7%	1502	37.4%	1441	39.9%	1387	42.2%

Tables 6 and 7 show the replacement word requirements for policy *b* when 10%, 20% and 40% of the scan cells are left in their original positions for 32 and 64 bit replacement word size respectively. For our experiments we randomly chose a number of different combinations of scan cells that are left in their original positions so as to examine how our method behaves in this case. When the constraint is rather relaxed, that is just 10% of the scan cells remain at their original positions the test data volume savings are comparable with the case where all scan cells are rearranged. Even if the constraint becomes more stringent, i.e. 40% of the scan cells are not moved from their original positions, the proposed method provides considerable savings in test data volume when compared with the method proposed in [19]. If we compare Tables 6 and 7 we observe the same behavior in terms of test data volume savings when the replacement word size increases as in the case of Tables 4 and 5.

Table 6. 32 bit replacement word requirements when a percentage of the scan cells is not moved

		32 bit replacement words with percentage of cells not moved					
Circuit	[19]	10%	Savings	20%	Savings	40%	Savings
s5378	632	399	36.9%	419	33.7%	486	23.1%
s9234	914	679	25.7%	736	19.5%	780	14.7%
s13207	986	786	20.3%	848	14.0%	880	10.8%
s15850	1163	954	18.0%	999	14.1%	1062	8.7%
s38417	5556	4061	26.9%	4434	20.2%	4703	15.4%
s38584	3995	2813	29.6%	3011	24.6%	3226	19.2%

Table 7. 64 bit replacement word requirements when a percentage of the scan cells is not moved

		32 bit replacement words with percentage of cells not moved					
Circuit	[19]	10%	Reduction	20%	Reduction	40%	Reduction
s5378	374	246	34.2%	269	28.1%	293	21.7%
s9234	575	432	24.9%	470	18.3%	518	9.9%
s13207	711	545	23.3%	562	21.0%	597	16.0%
s15850	761	592	22.2%	623	18.1%	678	10.9%
s38417	3252	2302	29.2%	2529	22.2%	2719	16.4%
s38584	2398	1534	36.0%	1687	29.6%	1924	19.8%

Finally we studied the case where the scan chain is considered as a succession of clusters of scan cells where we can rearrange the scan cells within each cluster but we cannot move a scan cell from one cluster to another (policy c). We can thus alleviate the negative effect of the scan cell rearrangement on area and/or timing constraints. We assumed that each cluster contains the scan cells whose respective bit columns are contained in two successive blocks of the test vector. Within each cluster we ordered

Table 8. Test data volume savings when column ordering happens in clusters of scan cells

	32 bit replacement words			64 bit replacement words		
Circuit	[19]	Proposed	Savings	[19]	Proposed	Savings
s5378	632	447	29.3%	374	282	24.6%
s9234	914	707	22.6%	575	429	25.4%
s13207	986	797	19.2%	711	555	21.9%
s15850	1163	953	18.1%	761	581	23.7%
s38417	5556	4179	24.8%	3252	2437	25.1%
s38584	3995	3050	23.7%	2398	1734	27.7%

the scan cells according to the number of transitions observed in the respective bit columns. Obviously some of the scan cells may have the constraint of staying in the same position in the cluster and therefore the ordering process must take this constraint into account. In the results presented in Table 8 we assumed that all the scan cells of each cluster can be moved anywhere in the cluster. Table 8 shows that considerable savings in test data volume can be obtained which reach up to 29.3% and 27.7% for replacement word size equal to 32 and 64 bits.

5 Conclusions

We have presented a new embedded processor based technique using test vector ordering and scan cell rearrangement for achieving significant reduction in the test data storage requirements of the tester with respect to already proposed techniques. The reduction in test data storage can reach up to 87.3% when compared with the initial test data size and up to 42.2% when compared with an already existing scheme. We also assessed the impact on test data volume reduction of different policies for the rearrangement of the scan cells. Furthermore since the proposed scheme uses the same approach for generating and applying test vectors as [19] it exhibits the same advantages, namely reduction in test time and the possibility of using already existing ATE equipment. The use of an existing processor in conjunction with ATE equipment can tackle with the increasing test demands posed by recent advances in the silicon manufacturing process.

References

1. Zorian, Y., "Test Requirements for Embedded Core-Based Systems and IEEE P1500", Proc. Intl. Test Conf., 1996, pp. 191-199.
2. Iyengar, V., Chakrabarty, K., Murray, B. T., "Deterministic Built-In Self Test of Sequential Circuits Using Precomputed Test Sets", Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 15, Aug./Oct. 1999, pp. 97-114.
3. Jas, A., Ghosh-Dastidar, J., Touba, N. A., "Scan Vector Compression / Decompression Using Statistical Coding", Proc. IEEE VLSI Test Symp., 1999, pp. 114-120.
4. Jas, A., Touba, N. A., "Test Vector Decompression Via Cyclical Scan Chains and Its Application to Testing Core-Based Designs", Proc. Intl. Test Conf., 1998, pp. 458-464.
5. Chandra, A., Chakravarty, K., "Test Data Compression for System-on-a-Chip Using Golomb Codes", Proc. VLSI Test Symp., 2000, pp. 113-120.
6. Chandra, A., Chakravarty, K., "System-on-a-Chip Test Data Compression and Decompression Architectures Based on Golomb Codes", IEEE Trans. on CAD, Vol. 20, No. 3, March 2001, pp. 355-368.
7. Jas, A., Ghosh-Dastidar, J., Ng , M.-E., Touba, N. A., "An Efficient Test Vector Compression Scheme Using Selective Huffman Coding", IEEE Trans. on CAD, Vol. 22, No. 6, June 2003, pp. 797-806.
8. Chandra, A., Chakravarty, K., "Test Data Compression and Test Resource Partitioning for System-on-a-Chip Using Frequency-Directed Run-Length (FDR) Codes", IEEE Trans. on Computers, Vol. 52, No. 8, Aug. 2003, pp. 1076-1088.

9. Gonciari, P., Al-Hashimi, B. M., Nicolici, N., "Improving Compression Ratio, Area Overhead, and Test Application Time for Systems-on-a-Chip Test Data Compression/ Decompression", Proc. Design, Automation and Test in Europe (DATE), 2002, pp. 604-611.
10. Krishna, C. V., Touba, N. A., "Reducing Test Data Volume Using LFSR Reseeding with Seed Compression", Proc. Intl. Test Conf., 2002, pp. 321-330.
11. Bayraktaroglu, I., Orailoglu, A., "Concurrent Application of Compaction and Compression for Test Time and Data Volume Reduction in Scan Designs", IEEE Trans. on Computers, Vol. 52, No. 11, Nov. 2003, pp. 1480-1489.
12. Koenemann, B., Barnhart, C., Keller, B., Snethen, T., Farnsworth, O., Wheater, D., "A SmartBIST Variant with Guaranteed Encoding", Proc. Asian Test Symp., 2001, pp. 325-330.
13. Rajski, J., Tyszer, J., Kassab, M., Mukherjee, N., "Embedded Deterministic Test", IEEE Trans. on CAD, Vol. 23, No. 4, May 2004, pp. 776-792.
14. Saxena, J., Ploicke, P., Cyr, K., Benavides, A., Malpass, M., "Test Strategy for TI's TMS320AV7100 Device", IEEE Intl. Workshop on Testing Embedded Core-Based Systems, 1998.
15. Rajsuman, R., "Testing a System-On-a-Chip with Embedded Microprocessor", Proc. Intl. Test Conf., 1999, pp. 418-423.
16. Hellebrand, S., Wunderlich, H.-J., Hertwig, A., "Mixed-Mode BIST Using Embedded Processors", Proc. Intl. Test Conf., 1996, pp. 195-204.
17. Dorsch, R., Wunderlich, H.-J., "Accumulator Based Deterministic BIST", Proc. Intl. Test Conf., 1998, pp. 412-421.
18. Papachristou, C. A., Martin, F., Nourani, M., "Microprocessor Based Testing for Core-Based System on Chip", in Proc. 36th Design Automation Conf., 1999, pp. 586-591.
19. Jas, A., Touba, N. A., "Deterministic Test Vector Compression/ Decompression for Systems-on-a-Chip Using an Embedded Processor", Journal of Electronic Testing: Theory and Applications (JETTA), Vol. 18, Issue 4/5, Aug. 2002, pp. 503-514.
20. Sinanoglu, O., Bayaktaroglu, I., Orailoglu, A., "Scan Power Reduction Through Test Data Transition Frequency Analysis", Proc. of the Intl. Test Conf., 2002, pp. 844-850.
21. Bonhomme, Y., Girard, P., Landrault, C., Pravossoudovitch, S., "Power Driven Chaining of Flip-flops in Scan Architectures", Proc. of the Intl. Test Conf., 2002, pp. 796-802.
22. Bonhomme, Y., Girard, P., Guiller, L., Landrault, C., Pravossoudovitch, S., "Efficient Scan Chain Design for Power Minimization During Scan Testing Under Routing Constraint", Proc. Intl. Test Conf., 2003, pp. 488-493.
23. Bellos, M., Bakalis, D., Nikolos, D., "Scan Cell Ordering for Low Power BIST", in Proc. of IEEE ISVLSI 04, 2004, pp. 281-284.
24. Lee, H. K., Ha, D. S., "ATALANTA: An efficient ATPG for combinational circuits", Dept. of Elect. Eng., Virginia Polytechnic Inst. and State Univ., Blacksburg, VA, USA, Tech Rep. 93-12, 1993.
25. Hamzaoglu, I., Patel, J. H., "Test Set Compaction Algorithms for Combinational Circuits", in Proc. ICCD '98, 1998, pp. 283-289.

Efficient Single-Pattern Fault Simulation on Structurally Synthesized BDDs

Jaan Raik, Raimund Ubar, Sergei Devadze, and Artur Jutman

Tallinn University of Technology, Department of Computer Engineering,
Raja 15, 12618 Tallinn, Estonia
{jaan, raiub, serega, artur}@pld.ttu.ee
http://ati.ttu.ee/english.php

Abstract. Current paper proposes an efficient alternative for traditional gate-level fault simulation. The authors explain how Structurally Synthesized Binary Decision Diagrams (SSBDD) can be used for representation, simulation and fault modeling of digital circuits. It is shown how the first phase of any fault simulation algorithm: the fault-free simulation can be accelerated using this model. Moreover, it is pointed out that simultaneous to simulation on SSBDDs, the set of potential fault locations can be significantly reduced. In addition, algorithms for deductive and concurrent fault simulation on SSBDD models are introduced in the paper. While full implementation of the new SSBDD based algorithms needs to be carried out, the paper presents experimental data revealing the advantages of the proposed data structure in the fault simulation process.

1 Introduction

Fault simulation is a widely used procedure in the digital circuit design flow. Most of the dependable computing and test design tasks (fault injection, test pattern generation, built-in self-test, etc.) rely on it. Thus, accelerating the subtask of fault simulation would consequently improve all the above-mentioned applications. In this paper we consider speeding up single-pattern fault simulation, for cases where pattern parallelism cannot be exploited (e.g. fault simulation with fault dropping). Traditionally, single-pattern fault simulation has been based on concurrent [1] or deductive [2] approaches.

The research topic of fault simulation is very mature. A lot of approaches and efficient circuit models have been proposed in the past. In [3], the logic netlist is partitioned into supergates. In other works, fault simulation is carried out for circuit lines corresponding to the fanouts [4]. Current paper combines the two above ideas and goes much further by introducing the circuit model of Structurally Synthesized Binary Decision Diagrams (SSBDD) for fault simulation. SSBDD models allow considerably coarser partitioning than supergates. Furthermore, the approach allows to restrict the number of considered circuit lines to the fanout points while allowing to work on the higher level of abstraction, without any need to descend to simulation at the gate-level.

Additional benefit of SSBDDs lies in the fact that it provides for a fault collapsing in the model itself without a need to specify the list of collapsed faults explicitly. In all the nodes of an SSBDD, both, stuck-at 0 and stuck-at 1 faults have to be covered in order to guarantee 100 % fault coverage of the corresponding gate-level circuit. This means that, contrary to traditional gate-level collapsing techniques, it is not necessary to check if a fault at a circuit line is included to the collapsed list. This in turn results in time savings during the simulation process.

The paper is organized as follows. Section 2 introduces the circuit model of SSBDDs for representing logic circuits. Section 3 explains how logic circuit simulation can be accelerated using SSBDDs. It is also shown how, simultaneously with SSBDD simulation, a restricted set of circuit lines with potential fault positions is extracted. In Section 4, SSBDDs are categorized by the nature of reconvergent fanout paths. Sections 5 and 6 present new, efficient fault simulation algorithms based on the deductive and concurrent approaches, respectively. Section 8 provides preliminary experimental results and Section 8 concludes the paper.

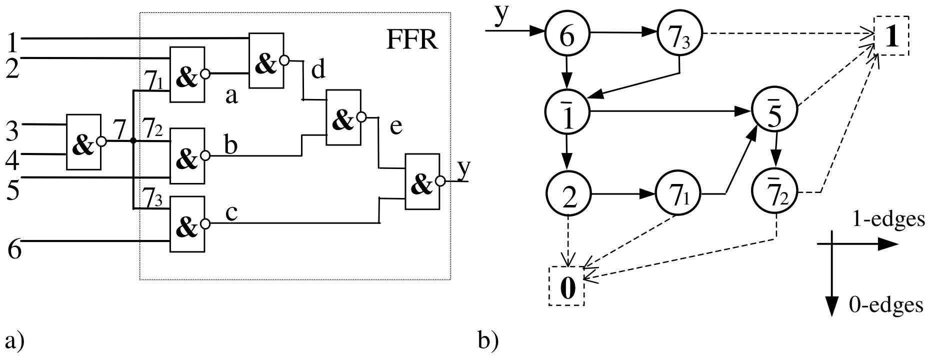


Fig. 1. a) Logic circuit and b) its SSBDD

2 Structurally Synthesized BDDs

Structurally Synthesized Binary Decision Diagram (SSBDD) [5-7] is a planar, acyclic BDD that is obtained by superposition of elementary BDDs for logic gates. SSBDDs were first introduced in [5]. The most significant difference between the traditional BDD and the SSBDD representations is the method how they are generated. While traditional BDDs are generated by Shannon's expansions, which extracts the function of the logic circuit, SSBDD models are generated by a superposition procedure that extracts both, function and data about structural paths of the circuit. Another difference between the classical and the SSBDD approach is that in SSBDDs we represent a digital circuit as a system of BDDs, where for each fanout-free region (FFR) of the circuit a separate SSBDD is generated.

An SSBDD G is a triple (M, X, Γ), where M is a set of nodes, $X(m)$ is a function, which defines line variables labeling the node m and $\Gamma(m, e)$ is a function, which gives the successor node of m with $X(m)=e$, $e\in\{0, 1\}$. The set of nodes M is divided into a set of nonterminal nodes $M_N=\{m_0,..,m_{k-1}\}$ and to a set of terminals M_T including **0**- and **1**-terminals, where $M = M_N \cup M_T$.

SSBDD models for gate-level digital circuits are created as follows. Starting from the output of the FFR (i.e. primary output or a fanout stem), logic gates are recursively substituted by their respective elementary BDDs. The procedure of superposition terminates in those nodes, which represent a primary input or a fanout branch.

Figure 1 shows an FFR of a logic circuit with an output line *y* and its corresponding SSBDD. Note, that in the Figure 1 we have omitted the 0- and 1-labels at the edges, since their direction (down or right) replaces the corresponding label. Downward edges correspond to 0-edges and rightward edges correspond to 1-edges. **0**- and **1**-terminal nodes are illustrated by dashed lines and can be also omitted. In the latter case, exiting the BDD downwards corresponds to *y*=0 and rightwards to *y*=1, respectively. In addition, SSBDD nodes can also be labeled by inverted variables (e.g. $\overline{1}, \overline{5}$, and $\overline{7}_2$ in Figure 1).

In order to consider simulation on SSBDD models, let us introduce some basic definitions. Let us denote $\Gamma(m, e)$ by m^e. Then m^0 is the successor of *m* for the value $X(m) = 0$ and m^1 is the successor of *m* for the value $X(m) = 1$. By the value assignment $X(m) = e$, $e \in \{0,1\}$, we say that the edge between nodes *m* and m^e is *activated.* Consider a situation where all the variables $X(m)$ are assigned by a Boolean vector $X^t \in \{0,1\}^n$ to some value. The edges activated by X^t form an *activated path* $l = (m_0, ..., m^T)$ from the root node m_0 to one of the terminal nodes $m_T \in M_T$.

In [7], it has been proved that G_y represents a Boolean function $y=f(X)$, and for all the possible vectors $X^t \in \{0,1\}^n$ a path $l = (m_0, ..., m_T)$ is activated so that $y=f(X^t)= x(m_T)$.

Definition. *High-path* (*low-path*) is the path $l = (m_1, \dots , m_L)$, where $m_{i+1} = m_i^1$ ($m_{i+1} = m_i^0$) for every $i : 1 \le i < L$.

Definition. Node *m* in the SSBDD is a *final node* if m^1 = **1**-terminal and m^0 = **0**-terminal.

The following theorem characterizes some simple properties of SSBDDs. The proof for the properties is provided in [7].

Theorem 1. If *G* is an SSBDD, then:

1. *G* is a planar graph.
2. There exists a high-path from every node to the **1**-terminal.
3. There exists a low-path from every node to the **0**-terminal.
4. *G* has exactly one final node.
5. There exists a directed path through all nonterminal nodes.
6. For every pair of nonterminal nodes m_1, m_2 there exists a directed path from m_1 to m_2 or from m_2 to m_1.

The above properties are the basis of the fault simulation acceleration ideas presented in Sections 3 to 5. Proof for the theorem has been published in [7].

Differently from traditional BDDs, SSBDDs support test generation for gate-level structural faults in terms of signal paths without representing these faults explicitly. Furthermore, the worst case complexity and memory requirements for generating SSBDD models for FFRs are linear in respect to the number of logic gates in the circuit, while for traditional BDDs the total storage space exceeds 2^n bits for an n-input combinational circuit [8]. Hence, SSBDDs for an arbitrary realistic-sized digital circuit can be generated very rapidly using only a small amount of computer memory.

3 Simulation on SSBDD Representations

The main contributions of the paper is the study of fault-free simulation on SSBDD models. Fault-free simulation is the initial step of any fault simulation algorithm.

The proposed algorithm is based on two basic properties of SSBDD models:

1) We previously saw that any Boolean vector $X^t \in \{0,1\}^n$ activates a path $l = (m_0, \ldots, m_T)$ from the root node m_0 to one of the terminal nodes $m_T \in M_T$ in an SSBDD G_y. Thus, faults at nodes which do not belong to the main activated path l cannot influence the value of y calculated by G_y with the selected vector X^t. Furthermore, it can be shown that faults at only those nodes mi, where X(mi)=f(Xt) can potentially influence the output. We call such faults fault candidates or *potential fault locations.*
2) Logic value for every primary output line y_O of a circuit with a Boolean vector X^t $\{0,1\}^n$ can be calculated by activating a path $l = (m_0, \ldots, m_T)$, where values $X(m_i)$ that do not correspond to primary outputs are calculated recursively. Obviously, faults in SSBDDs that are not traversed in such a recursion cannot change the value of y_O.

Properties 1 and 2 are illustratedin Figure 2. SSBDD model in that figure decribes a combinational circuit that consists of 7 FFRs, thus, represented by a model of 7 SSBDDs. The bold arrows indicate activated edges in each SSBDD simulated in a recursive manner starting from the SSBDD corresponding to the output line y. Recursive simulation starts from SSBDD G_y, traverses nodes a, c, d, f and then recursively simulates corresponding SSBDDs G_a, G_c, G_d and G_f. As a result, basing on the above properties, only 8 nodes out of 17 are identified as potential fault locations in this recursive simulation example. Note, that there was no need to simulate SSBDDs for b and e for finding the fault locations. However, in order to carry out fault propagation these graphs have to be simulated as well.

These simple properties are also applied in Algorithm 1, which both, marks the nodes containing potential faults, and performs full fault-free simulation. As the first step of the algorithm, primary outputs of the design are recursively simulated on the SSBDD representation in order to identify SSBDDs in which faults could be activated. Then, the rest of the SSBDDs will be simulated.

As the experiments carried out in Section 7 show, recursive simulation on SSBDDs is generally faster than simulation on the gate-level representations. What is more important, SSBDD simulation allows to considerably minimize the list of circuit lines to be regarded as potential faults in further fault simulation. The improvement achievable by the proposed SSBDD simulation is twofold:

1) SSBDD models provide implicit fault collapsing. The list of potential fault locations is reduced in average to 66 % of the uncollapsed one (See Section 7). While there exist more agressive fault collapsing approaches, this reduction has to be taken into account.

2) As experiments in Section 7 show, Algorithm 1 narrows the circuit lines with potential fault locations to less than 40 %

Algorithm 1. Fault-free simulation and identification of potential faults

```
FaultfreeSimulation()
{   for each primary output $y_O$
        RecursivelySimulate($y_O$)
    end for

    for each SSBDD $G$
        if $G$ is not marked as simulated then
            simulate $G$
        end if
    end for
}

RecursivelySimulate($y$)
{   if $y$ is a primary input then
        return the value of $y$.
    else
        mark $G_y$ as simulated
        $m = m_0$
        while $m \notin M_T$
            if $G_{X(m)}$ is marked as simulated then
                $m = m^{X(m)}$
            else
                $e$ = RecursivelySimulate($X(m)$)
                $m = m^e$
            end if
        end while
    end if
}
```

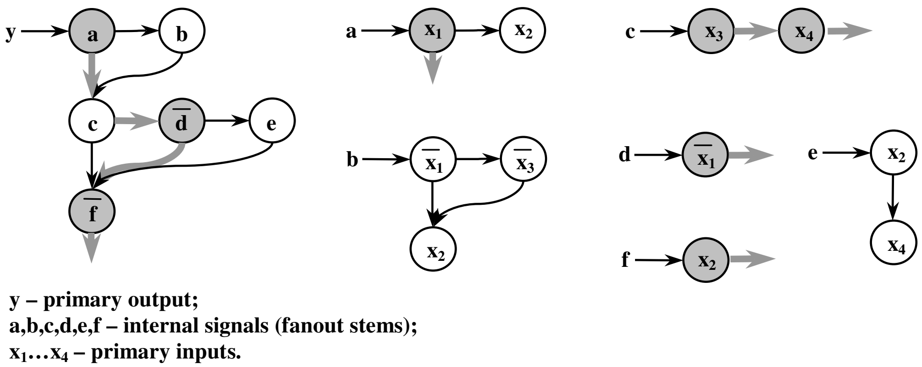


Fig. 2. Recursive simulation and fault candidate identification on SSBDDs

In addition to that, only one of two stuck-at faults per location can be active at a time. Thus, the SSBDD simulation proposed in Algorithm 1 reduces the set of faults to be simulated to $0.66 \cdot 0.4 \cdot 0.5 = 0.13$. In other words, in the fault simulation phase we have to operate with only 13 percent (!) of faults from the entire fault list.

4 Circuit Structure Analysis for SSBDD Fault Propagation

Efficient fault simulation methods carry out fault simulation for the list of faults within a Fanout-Free Region (FFR) of a circuit and subsequently replace this list by the fault at the output of the FFR [4]. The fault at the fanout stem will then represent all the faults inside the corresponding FFR. This approach is very well supported in SSBDD models, where FFRs and SSBDDs have one-to-one correspondence. It provides for minimization of the size of the fault list, which is crucial in accelerating single-pattern fault simulation, since the speed of deductive and concurrent fault simulation algorithms greatly depends on the speed of set operations on fault lists.

Current paper introduces additional means for improving fault simulation by applying structural analysis. We propose that the FFRs (i.e. SSBDDs) of the circuit should be categorized into three categories by the nature of reconverging fanout paths in them:

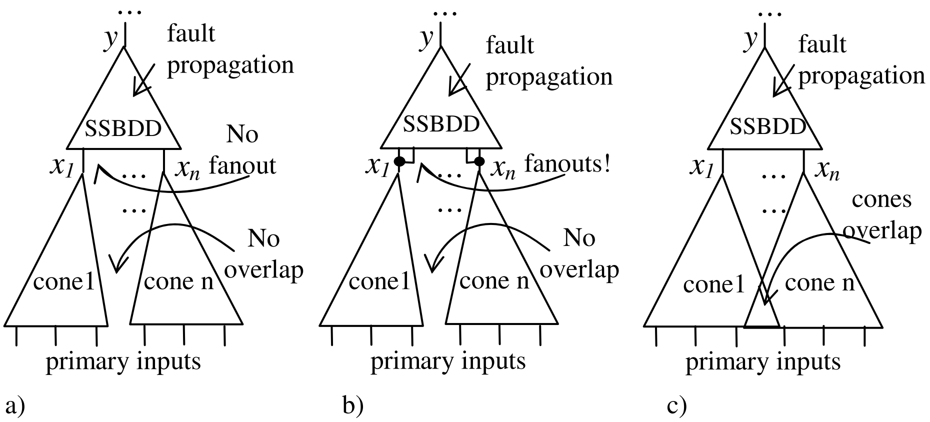


Fig. 3. SSBDDs a) w/o reconvergencies, b) with reconvergencies of depth 1 and c) with complex reconvergencies of depth > 1

1. FFRs with no reconvergent paths;
2. FFRs with reconvergent paths with directly preceding FFRs (i.e. maximum reconvergence depth = 1);
3. FFRs with arbitrary reconvergencies (i.e. max. reconvergence depth > 1).

Figure 3 explains that concept for an SSBDD G_y containing nodes with variables x_1, ..., x_n.

Table 1. Distribution of SSBDDs according to reconvergencies

circuit	total SSBDDs	SSBDDs w/o reconv.	max reconv. depth = 1	max reconv. depth > 1
c17	5	5	0	0
c432	96	95	1	0
c499	187	123	64	0
c880	151	133	18	0
c1355	291	187	40	64
c1908	248	180	68	0
c2670	430	391	24	15
c3540	378	344	31	3
c5315	633	529	91	13
c6288	1488	1488	0	0
c7552	920	736	160	24

In the following Sections we will present a fault simulation algorithm that takes advantage of the above-mentioned classification. Fault propagation through an SSBDD with no reconvergencies can be done simultaneously to fault activation. In order to propagate faults through an SSBDD with reconvergent paths with maximum depth 1, faults in variable values of $X(m)$ have to be additionally simulated (See Algorithm 2). Finally, for SSBDDs with a more complex reconvergency structure we can resort to the deductive fault propagation method described in Algorithm 3.

Table 1 shows such classification of SSBDDs according to reconvergency properties for the ISCAS'85 circuits. As we can see, in most of the circuits there are no reconvergencies with depths exceeding one FFR. The average percentage of such FFRs is only 2.5 %. The only circuit with a large portion of complex reconvergencies is c1355, where 22 % of all the FFRs fall to the third category.

5 Concurrent Fault Simulation

Concurrent fault simulation [1] can be viewed as an extension to event-driven fault simulation. Lists of activated faults are propagated concurrently through the circuit and faulty circuit events are simulated. In the following, a concurrent fault simulation algorithm for SSBDD representations is proposed. Let us start with notations.

For each SSBDD G_y the following fault simulation algorithm is implemented. First, all the nodes belonging to M_1 are traversed. If a node corresponds to a line of fanout reconvergence then the variable labeling the node is changed to a faulty value and G_y is simulated. Otherwise, fault is injected at the node and simulation is performed. Fault injection and simulation is used also for determining the faults activated in G_y in order to compute the set A. If A will be an nonempty set then after the simulation y *stuck-at*-$\urcorner f(X^t)$ will represent all the faults in A because it is dominated by them for the vector X^t.

Algorithm 2. Concurrent fault simulation on SSBDDs.

```
ConcurrentFaultSimulation()
{
  $R = \emptyset, A = \emptyset$
  for each $m \in M_1$
      FaultSimulateNode($m$)
      if $S(m)$ contains faults reconverging in $m$ then
            $X(m) = \urcorner X(m)$
            simulate $G_y$
            if the faulty value of $y$ differs from the fault-free one then
                  $R = R \cup S(m)$
            end if
            $X$(m) = $\urcorner X(m)$
      else
            if fault at node $m$ is mrked as active inside the FFR
                  $R = R \cup S(m)$
            end if
      end if
  end for
}

FaultSimulateNode($m$)
{
    follow the activated path $l = (m^{\urcorner X(m)}, ..., m_T)$
    if($X(m_T) = \urcorner f(X^t)$) then
        $A = A \cup \{$ $m$-$stuck$-$at$-$\urcorner X(m)$ $\}$
        mark fault at node $m$ as active inside the FFR
    end if
}
```

Notations:

- M_T - set of terminal nodes (i.e. 0-terminal and 1-terminal).
- M_N - set of nonterminal nodes.
- M_1 - set of nonterminal nodes at the main activated path l_{main}, where $\{ m \in M_1 | X(m)=f(X^t) \}$
- $m' = m^{X(m)}$ $m'' = m^{\urcorner X(m)}$
- $S(m)$ – fault list propagated to m from previous SSBDDs.
- A – the set of faults active in the FFR.
- R – the set of faults propagated to circuit line corresponding to the FFR output y.

Note, that the method described by Algorithm 2 can only be applied to graphs, whose reconvergencies are of depth 1 or less (See Section 3). For SSBDD graphs with no reconvergent fanout paths, the check for reconvergencies can be omitted.

6 Deductive Fault Effect Propagation

In deductive fault simulation [2], signal values for the fault-free circuit are computed and faults that will cause each line to have a value different from its fault-free value are deduced. In the following algorithm the computation is performed on SSBDDs based on the properties stated by theorems 1 and 2.

Notations:

- M_T - set of terminal nodes (i.e. 0-terminal and 1-terminal).
- M_N - set of nonterminal nodes.
- M_1 - set of nonterminal nodes at the main activated path l_{main}, where $\{ m \in M_1 \mid X(m)=f(X^t) \}$
- $m' = m^{X(m)}$ $m'' = m^{\neg X(m)}$
- $S(m)$ – fault list propagated to m from previous SSBDDs.
- $L(m)$ – temporary fault list for calculating the propagated faults R.
- R – the set of faults propagated to circuit line corresponding to the FFR output y.

Algorithm 3. Deductive fault propagation on an SSBDD.

```
DeductivePropagation()
{
   $R = \emptyset$
   for each $m \in M_N$
      $L(m) = \emptyset$
   end for
   for each $m \in M_N$
      if $m \in M_1$ then
         if $m'' \notin M_T$ then
            $L(m'') = L(m'') \cup S(m)$
         else
            $R = R \cup L(m)$
         end if
      else
         if $m' \notin M_T$ then
            $L(m') = L(m') \cup (S(m) \setminus L(m))$
         else if $X(m) = \neg f(X^t)$ then
            $R = R \cup L(m)$
         end if
         if $m'' \notin M_T$ then
            $L(m'') = L(m'') \cup (S(m) \cap L(m))$
         else if $x(m) = \neg f(X^t)$ then
            $R = R \cup L(m)$
         end if
      end if
   end for
}
```

The task of the presented deductive fault propagation algorithm is to calculate the set of faults *R* propagated through the SSBDD G_y by a given input vector X^t and *S(m)*. Note, that the proposed algorithm does not include fault activation. This can be carried out in the similar manner as described in Algorithm 2.

One of the main shortcomings of the deductive approach is relatively time-consuming set operations on long fault lists. However, as experiments show, SSBDD models allow to significantly reduce the list of faults to be considered. Furthermore, as circuit structure analysis presented in Section 4 showed the number of graphs where deductive fault simulation is usually very small.

Finally, Algorithm 4 combines the ideas proposed in this paper into a single fault simulation approach.

7 Experimental Results

A useful property of SSBDDs is its ability to provide fault collapsing in the model. While in the gate-level descriptions we model stuck-at faults at the interconnections between the gates, in SSBDD representations the faults are present at nodes. For example, stuck-at-0 fault at a node is modeled with the 0-edge of the node being constantly activated, regardless of the value of the variable labeling this node. Each SSBDD node represents a distinct path in the corresponding fanout-free circuit. By testing all the SSBDD node faults we will consequently test all the signal paths in the circuit and thus all the single stuck-at faults. This ability of SSBDDs to implicitly model logic level stuck-at faults is a very important property, which distinguishes it from other classes of BDDs.

Algorithm 4. Fault simulation on SSBDD models.

```
SSBDDfaultSimulation()
{
    Recursively simulate SSBDD model and
      identify the list of potential faults    // Algorithm 1

    for each SSBDD G
      if G has no reconvergent paths then
          Simultaneous fault activation and fault propagation
          // See Algorithm 2!
      else if G has reconvergent paths with max. depth = 1
          ConcurrentFaultSimulation()    // Algorithm 2
      else
          Fault activation in SSBDD      // FaultSimulateNode in Alg. 2
          DeductivePropagation()         // Algorithm 3
      end if
    end for
}
```

Table 2 compares the number of uncollapsed faults, the number of standard collapsed faults and the number of SSBDD faults in the ISCAS'85 benchmark set. As we can see that reduction provided by collapsing in the SSBDD model is about 1.5 times.

While there exist more aggressive fault collapsing approaches the advantage of the SSBDD based collapsing over the traditional one is that it allows us at the same time to rise to a higher abstraction level of circuit modeling. In the traditional case we would only minimize the number of faults but would still be working at the level of logic gates.

Table 3 presents the fault-free simulation results, which were carried out to evaluate the speed of SSBDD simulation and its ability to restrict the number of simulated faults (See Algorithm 1). The simulation algorithms were compiled by GNU C compiler using –O option and experiments were run on a 366MHz SUN Ultra60 server using SUN Solaris 2.8 operating system.

Table 2. Number of collapsed and SSBDD faults in ISCAS85 circuits

circuit	Faults	collapsed	SSBDD
c880	1550	**942**	994
c1355	2194	**1574**	1618
c1908	2788	1879	**1732**
c2670	4150	2747	**2626**
c3540	5568	3428	**3296**
c5315	8638	**5350**	5424
c6288	9728	**7744**	**7744**
c7552	11590	7550	**7104**

Table 3. SSBDD simulation experiments

circuit	vectors	Gate	SSBDD		SSBDD density, %	node density, %
		simulation, s	simulation, s	recursive, s		
c432	100000	3.76	1.56	2.12	66.45	30.80
c499	10000	1.06	0.33	0.55	94.74	46.84
c880	10000	0.70	0.27	0.44	89.83	44.74
c1355	10000	0.97	0.58	0.89	93.73	52.33
c1908	10000	1.28	0.50	0.75	81.13	37.54
c2670	10000	2.13	0.85	1.16	80.55	38.09
c3540	10000	2.74	0.87	1.21	76.38	30.27
c5315	1000	0.42	0.17	0.21	67.45	27.92
c6288	1000	0.49	0.34	0.52	100	58.30
c7552	1000	0.57	0.25	0.34	79.82	37.37

The second column of the table shows the number of simulated vectors. The third column shows the time spent by gate-level simulation, in seconds. The fourth column presents the time needed to simulate all the SSBDDs in the model. The fifth column shows the time for recursive SSBDD based simulation as shown in Algorithm 1. The run times in the fifth column include identification of the potential faults to be simulated.

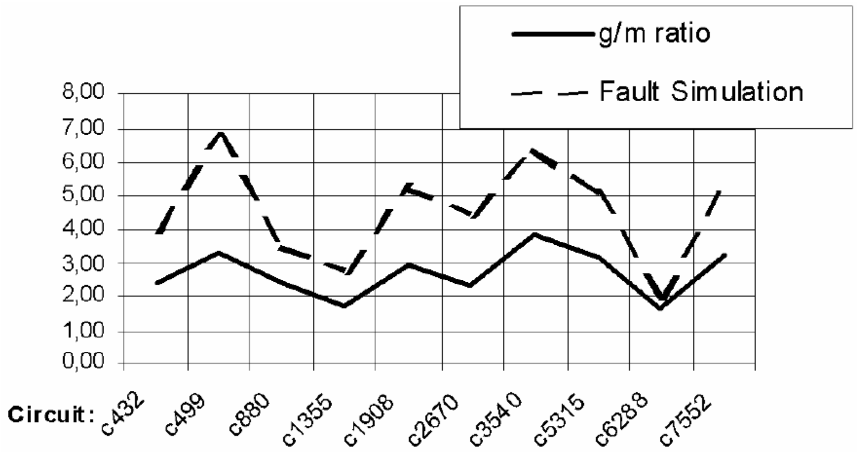


Fig. 4. Fault simulation speed-up and average SSBDD/gate ratio

The two last columns reflect the ability of recursive SSBDD simulation to reduce the set of considered faults. ‚SSBDD density' shows the percentage of SSBDDs traversed by recursive simulation from the total number of FFRs. The column ‚Node density' the percentage of SSBDD nodes identified as fault candidates by Algorithm 1. The average SSBDD density was 80 % and node density 40 %. In the worst case, c6288 (a parallel multiplier), SSBDD density was 100 %, i.e. all the SSBDDs were traversed by recursive simulation, and the node density was 58.3 %. For all the benchmarks except c6288 Algorithm 1 performed faster than the fault-free simulation at the gate-level.

Figure 4 presents comparison of implementation of the event-driven fault simulation algorithms on SSBDDs and logic gates. We can see from the Figure that the speedup of SSBDD fault simulation ('Fault Simulation') ranges from 2 to 7 times and that it is well correlated with the average number of logic gates in an FFR ('g/m ratio').

8 Conclusions and Future Work

The paper proposes an efficient alternative for traditional gate-level fault simulation based on Structurally Synthesized Binary Decision Diagrams (SSBDD). It is shown how fault-free simulation can be accelerated using this model. Moreover, it is pointed out that simultaneous to simulation on SSBDDs, the set of potential fault locations can be significantly reduced. Experiments show that this reduction is in average 13.3 % from the total fault list (i.e. 26.6 % of the circuit lines). In addition, new algorithms for deductive and concurrent fault simulation on SSBDD models are introduced in the paper and structural analysis allowing a trade-off of these algorithms is performed.

As a future work we plan to compare the speed of the proposed algorithm to existing state-of-the-art commercial software. For additional speed-up we plan to implement dynamic node reordering [9] to the SSBDD models to be simulated.

Acknowledgements

This work was supported by EU projects IST 2000-30193 REASON, IST-2001-37592 eVIKINGS II, Estonian Science Foundation grants 5637, 5649, 5910, and by Enterprise Estonia Competence Center of Mission Critical Embedded Systems (ELIKO).

References

1. E. G. Ulrich, T. Baker, “Concurrent Simulation of Nearly Identical Digital Networks”, Computer, vol. 7, pp. 39-44, Apr. 1974.
2. D. B. Armstrong, “A Deductive Method for Simulating Faults in Logic Circuits”, IEEE Trans. Comput., vol. C-21, pp. 464-471, May 1972.
3. S. C. Seth, L. Pan, V. D. Agrawal, “Predict-Probabilistic Estimation of Digital Circuit Testability”, Dig. Papers FTCS-15, June 1985, pp. 220-225.
4. K. J. Antreich, M. H. Schulz, “Accelerated Fault Simulation and Fault Grading in Combinational Circuits”, IEEE Trans. on CAD, vol. CAD-6, no. 5, Sept. 1987, pp. 704-712.
5. R. Ubar. “Test Generation for Digital Circuits Using Alternative Graphs”, in Proc. Tallinn Technical University, 1976, No.409, Tallinn TU, Tallinn, Estonia, pp.75-81.
6. R. Ubar, “Test Synthesis with Alternative Graphs,” IEEE Design & Test of Comp. Spring 1996, pp. 48-59.
7. A. Jutman, A. Peder, J. Raik, M. Tombak, R. Ubar. Structurally synthesized binary decision diagrams. 6th International Workshop on Boolean Problems, pp. 271-278, Freiberg, Germany, Sept. 23-24, 2004.
8. H.-T. Liaw, C.-S. Lin, “On the OBDD-representation of general Boolean functions”, IEEE Trans. on Computers, Vol. C-41, No. 6, pp. 61-664, June 1992.
9. R. Ubar, T. Vassiljeva, J.Raik, A.Jutman, M. Tombak, A. Peder. Optimization of Structurally Synthesized BDDs. Proc of the 4th IASTED International Conference on Modelling, Simuation, and Optimization MSO 2004, pp. 234-240, Kauai, Hawaii, USA, Aug. 17-19, 2004.

Structure-Based Resilience Metrics for Service-Oriented Networks

Daniel J. Rosenkrantz[1], Sanjay Goel[2], S.S. Ravi[1], and Jagdish Gangolly[3]

[1] Department of Computer Science,
University at Albany – State University of New York,
Albany, NY 12222
{djr, ravi}@cs.albany.edu

[2] Department of Management Science and Information Systems,
School of Business, University at Albany – State University of New York,
Albany, NY 12222
goel@albany.edu

[3] Department of Accounting and Law,
School of Business, University at Albany – State University of New York,
Albany, NY 12222
j.gangolly@albany.edu

Abstract. Many governmental agencies and businesses organizations use networked systems to provide a number of services. Such a service-oriented network can be implemented as an overlay on top of the physical network. It is well recognized that the performance of many of the networked computer systems is severely degraded under node and edge failures. The focus of our work is on the resilience of service-oriented networks. We develop a graph theoretic model for service-oriented networks. Using this model, we propose metrics that quantify the resilience of such networks under node and edge failures. These metrics are based on the topological structure of the network and the manner in which services are distributed over the network. Based on this framework, we address two types of problems. The first type involves the analysis of a given network to determine its resilience parameters. The second type involves the design of networks with a given degree of resilience. We present efficient algorithms for both types of problems. Our approach for solving analysis problems relies on known algorithms for computing minimum cuts in graphs. Our algorithms for the design problem are based on a careful analysis of the decomposition of the given graph into appropriate types of connected components.

1 Introduction

1.1 Motivation

Federal and state governmental agencies, businesses and other organizations use networked systems to provide a number of services. Private networks maintained by organizations allow one component of the organization (such as the Emergency Management Agency) to access services provided by other components (such as Transportation Agency, Health Services, etc.). In these situations, fast and reliable access to information is needed. Several papers in the literature have proposed architectures for **service-oriented networks** (see for example [6, 7, 8, 13]). In such an architecture, each node is associated with two sets of services, namely local and non-local services. When a user requests a service that is locally available at a node, the node provides the service directly. When the requested service is non-local, the node forwards the user's request to another node in the network where the service is available, and relays the response to the user. Different approaches for implementing service-oriented networks are available in the literature (e.g. [5, 8, 12, 13, 14, 18, 22, 26]). Typically, such networks are implemented as overlay networks on top of the physical network.

It is well recognized that many of the networked computer systems used by various government agencies and organizations are not resilient enough to withstand failures and attacks. The performance of these networks is severely degraded by failures. Thus, it is important to develop techniques for designing and implementing resilient service-oriented networks which can not only survive attacks and failures but also continue to provide a reasonable level of service to support critical infrastructure and activities of governmental agencies and organizations. Methods for implementing networks that can continue to function under node and link failures are known (see for example [12, 16, 22]). Also, techniques for data replication to ensure availability of data in the event of network failures are available (see for example [4, 9, 11, 16, 27]). In the context of resilient service-oriented networks, services must also be replicated. Further, when a failure or anomalous behavior is sensed by the network, it must have the ability to migrate services to other nodes across the network, taking into consideration the resource constraints at the nodes along the migration path. Such a feature is necessary to allow critical services to be maintained on the network during duress and is especially useful for state and federal agencies in handling crisis situations. Thus, effective methods for the design and analysis of resilient service-oriented networks must accommodate a variety of features and resource constraints.

1.2 Our Contributions

Much of the work in the literature on service-oriented and overlay networks has addressed issues such as the maintenance of routing tables at nodes and reliable transmission of service requests and network status information across the network. Our work considers the resilience of service-oriented networks at a higher level of abstraction, assuming that lower level mechanisms for basic network functions such as routing and service discovery are supported by the system. The focus of our work is on the identification of suitable metrics that can be used to quantify the resilience of service-oriented networks under node and edge failures. Such metrics are useful in assessing the reliability of a given network (i.e., in analyzing a given network) as well as in choosing an appropriate network topology and/or an optimal distribution of services over the network (i.e., in designing the network).

We propose a graph theoretic model for service-oriented networks and use this model to identify some resilience metrics. These metrics are based on the topology of the network and the manner in which services are distributed over the network. An example of such a metric is **node resilience**, which specifies the maximum number of node failures that a service-oriented network can tolerate and still continue to provide services to users. (Precise definitions of this and other metrics are given in Section 2.) These metrics are different from other notions of network resilience (e.g. the minimum number of nodes or edges whose failure will disconnect the network) studied in the literature (see for example [2, 15, 17, 19, 23, 24, 25]). In particular, our metrics for service-oriented networks explicitly consider the distribution of services over the nodes of a network. Additional discussion regarding these metrics is provided in Section 2.

Having identified some resilience metrics, we develop algorithms for analysis and design problems arising in the context of resilient service-oriented networks. Given a service-oriented network, the goal of the analysis problem is to compute the node and edge resilience parameters of the network. We develop polynomial time algorithms for these problems. These algorithms are derived through a transformation to the problem of computing minimal cutsets in graphs. The design problem addressed in this paper concerns the placement of services at the nodes of a given network so that the cost of placing the services is minimized and the resulting network has a specified level of resilience. We consider this problem for single node and single edge failures, and develop polynomial algorithms. These algorithms are based on a careful analysis of the decomposition of the given graph into appropriate types of connected components.

2 Formal Model and Structure-Based Resilience Metrics

2.1 Graph Model and Definitions of Metrics

Following standard practice [19, 25], we model a network as an undirected connected graph. Each node represents a computer system and each edge represents a bidirectional link between the corresponding pair of systems. To model service-oriented networks, we associate two sets of services with each node. For a node v, $A(v)$ denotes the set of services available locally at v, and $N(v)$ denotes the set of nonlocal services needed at v. In other words, node v supports each service in $N(v)$ by forwarding requests for such a service to one or more nodes that offers it as a local service. Thus, the sets $A(v)$ and $N(v)$ are disjoint. A user connected to node v may request any service in $A(v) \cup N(v)$. For a node v, if service $s \in N(v)$, we say that v is a **demand point** for service s.

The nodes and/or links of a network may fail either because of an attack or because of equipment failure. We assume that node failures correspond to system crashes. Thus, we do not consider Byzantine node failures. In general, a system that has crashed cannot communicate with any of its neighbors. Thus, each node failure can be modeled by deleting the failed node and all the edges incident on the failed node from the underlying graph of the network. When a link fails, it is assumed that no communication across the link is possible in either direction.

Thus, each link failure may be modeled by the deletion of the corresponding edge from the underlying graph. Using such models of node and edge failures, several network resilience metrics have been considered in the literature (see for example [1, 2, 15, 17, 19, 20, 21, 23, 24, 25]). Examples of such metrics include node and edge connectivity parameters of a graph (i.e., the minimum number of nodes or edges that must be removed to disconnect the graph). Such metrics characterize the ability of a network to remain connected when nodes and edges fail. They do not take into account the distribution of services across a network. As will be seen, a service-oriented network may continue to function even when the underlying graph is disconnected. Thus, new metrics are needed to capture the notion of resilience in service-oriented networks.

Node and link failures may partition a network into a collection of two or more connected components or subnetworks. In such a case, nodes in one subnetwork cannot access the services provided by the nodes in another subnetwork. We say that a subnetwork is **self-sufficient** if each service needed by a node in that subnetwork is provided by some node in the same subnetwork; otherwise, the subnetwork is said to be **deficient**. Thus, a self-sufficient subnetwork can continue to process requests from users even though it has been sequestered from the rest of the network due to node and link failures. Given a network $G(V, E)$ and a subset $S \subseteq V \cup E$ of nodes and/or edges, we say that the network G is **resilient** with respect to the failure set S if each of the subnetworks that results when all the nodes/edges in S fail is self-sufficient.

We can now define the structure-based resilience metrics proposed in this paper. To do this, we consider failure sets under two separate categories, namely node failures and edge failures. In the former, each failure set consists only of nodes, and in the latter, each failure set consists only of edges.

Definition 1. *(a) A service-oriented network is **k-edge-failure-resilient** if no matter which subset of k or fewer edges fails, each resulting subnetwork is self-sufficient. The **edge resilience** of a network is the largest integer k such that the network is k-edge-failure-resilient.*
*(b) A service-oriented network is **k-node-failure-resilient** if no matter which subset of k or fewer nodes fails, each resulting subnetwork is self-sufficient. The **node resilience** of a network is the largest integer k such that the network is k-node-failure-resilient.*

Example: Consider the eight node network shown in Figure 1. The eight services provided by the network are denoted by s_1 through s_8. For each node v_i , the figure also shows the sets A_i (the set of services available at v_i) and N_i (the set of services needed at v_i), $1 \leq i \leq 8$. Note that the node connectivity and the edge connectivity of the network are both one, since the network can be disconnected by removing one node (for example, the node v_3) or one edge (the edge $\{v_3, v_5\}$). However, the node and edge resilience parameters of the network are both two. In particular, the subnetworks obtained by deleting the edge $\{v_3, v_5\}$ or one of the nodes v_3 and v_5 are all self-sufficient. It can be verified that no matter which pair of vertices or which pair of edges is deleted, each of the resulting subnetworks is self-sufficient. However, when the three edges $\{v_1, v_2\}$, $\{v_1, v_3\}$

$A_1 = \{s_1, s_2, s_3\}$	$N_1 = \{s_4\}$
$A_2 = \{s_1, s_2, s_4\}$	$N_2 = \{s_3\}$
$A_3 = \{s_1, s_3, s_4\}$	$N_3 = \{s_2\}$
$A_4 = \{s_2, s_3, s_4\}$	$N_4 = \{s_1\}$
$A_5 = \{s_5, s_6, s_7\}$	$N_5 = \{s_8\}$
$A_6 = \{s_5, s_6, s_8\}$	$N_6 = \{s_7\}$
$A_7 = \{s_5, s_7, s_8\}$	$N_7 = \{s_6\}$
$A_8 = \{s_6, s_7, s_8\}$	$N_8 = \{s_5\}$

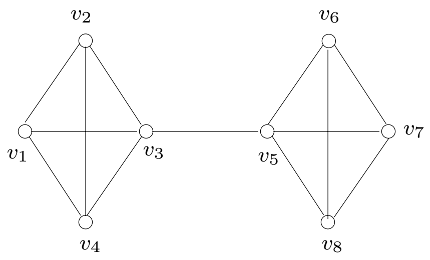


Fig. 1. Example to Illustrate Resilience Metrics

and $\{v_1, v_4\}$ are deleted, the subnetwork containing only the node v_1 is deficient, since it does not have access to service s_4. Likewise, when the three nodes v_1, v_2 and v_3 are deleted, the subnetwork containing only the node v_4 is deficient, since it does not have access to service s_1.

2.2 Problem Formulation

We now provide precise formulations of the analysis and design problems considered in this paper. We start with a specification of analysis problems.

Edge Resilience Problem (Erp)
Instance: A service oriented network consisting of an undirected graph $G(V, E)$ and the sets A_v and N_v for each node $v \in V$.

Requirement: Compute the edge resilience of the network.

The formulation of **Node Resilience Problem** (Nrp) is similar to that of Erp. Section 3 presents efficient algorithms for these two problems.

Many versions of design problems for resilient service-oriented networks can be formulated. We focus on design problems where the goal is to choose an optimal distribution of services over a given network. In other words, we assume that the network topology and the set of services needed at each node are given, and the objective is to find the set of services to be provided by each node so as to achieve a desired degree of node or edge resilience. In the absence of cost considerations, such problems can be solved trivially by having all the services at each node. In general, such solutions are economically infeasible. Moreover, some of the nodes in the network may not have the computational resources needed to support a certain service. Therefore, we assume that there is a cost associated with placing services at nodes and that the total cost of placing the services must be minimized. We can now provide a precise formulation of the design problems considered in this paper.

Design for Edge Resilience (Der)
Instance: An undirected graph $G(V, E)$, with $V = \{v_1, v_2, \ldots, v_n\}$, a set $S = \{s_1, s_2, \ldots, s_p\}$ of services, a set $N(v_i) \subseteq S$ for each node $v_i \in V$, an $n \times p$ cost matrix $C = [c_{ij}]$, where $c_{ij} \geq 0$ is a real number that denotes the cost of placing service p_j at node v_i , $1 \leq i \leq n$ and $1 \leq j \leq p$, and integer K.

Requirement: For each node $v_i \in V$, compute a set $A(v_i) \subseteq S$ of services to be placed at v_i, such that the edge resilience of the resulting service-oriented network is at least K and the total cost of placing the services is minimized.

A solution to the Der problem may place a service s_j at a node v_i when $s_j \in N(v_i)$. In such a case, the sets $A(v_i)$ and $N(v_i)$ for node v_i are no longer disjoint. To ensure that the two sets remain disjoint, one can modify the set $N(v_i)$ into a new need set $N'(v_i)$ by deleting the services in $N(v_i) \cap A(v_i)$.

The formulation of the Design for Node Resilience (Dnr) problem is similar, except that the parameter K represents the required level of node resilience.

In Section 4, we present polynomial algorithms for the Der and Dnr problems, assuming that the value of K is 1. Even for $K = 1$, the algorithms involve careful analyses of certain types of decompositions of undirected graphs. For larger values of node and edge resilience parameters, developing properties of appropriate graph decompositions appears to be a nontrivial task. So, we leave the design problems for higher resilience values as directions for future work.

3 Algorithms for Analyzing a Given Network

3.1 An Algorithm for Computing Edge Resilience

In this section, we present our algorithm for computing the edge resilience of a given service-oriented network. We begin with some definitions.

Let $G(V, E)$ denote the underlying graph of the given network. We use the term **subnetwork** to mean a connected subgraph of G. (Subnetworks may contain just a single node.) For each node v in a subnetwork H, the sets $A(v)$ and $N(v)$ are the same as those in G. For any subnetwork H and any service $s_j \in S$, we say that H is **deficient with respect to** s_j if there is a node in H that needs s_j and no node in H provides s_j. Thus, a subnetwork H is deficient[1] if there is a service s_j with respect to which H is deficient.

Definition 2. *Let $G(V, E)$ denote the underlying graph of a service-oriented network. Let S denote the set of all services available in the network.*

(a) Given a service $s_j \in S$, a set of edges $Q \subseteq E$ is called a ***deficiency inducing edge*** *(*Die*)* ***set*** *for G* ***with respect to service*** *s_j if at least one of the connected components of the graph $G'(V, E - Q)$ is deficient with respect to s_j.*

(b) A set of edges $Q \subseteq E$ is a ***deficiency inducing edge set*** *for G if at least one of the connected components of the graph $G'(V, E - Q)$ is deficient.*

A direct consequence of the above definitions is that if Q^* is a Die set of minimum cardinality for G, then the edge resilience of G is equal to $|Q^*| - 1$. Therefore, we focus on computing $|Q^*|$. Our approach for finding $|Q^*|$ relies on the following observation.

[1] The definition of a deficient subnetwork was presented in Section 2.1.

Observation 1. *Let $G(V, E)$ denote the underlying graph of a service-oriented network. Let S denote the set of all services available in the network. For each service $s_j \in S$, let σ_j denote the minimum cardinality of a* Die *set for G with respect to service s_j. Let σ^* denote the minimum cardinality of a* Die *set for G. Then, $\sigma^* = \min\{\sigma_j : 1 \le j \le |S|\}$.* $\square$

The above observation points out that the cardinality of a smallest Die set for G can be computed by considering each service separately. We now show that for any service s_j, the problem of computing a minimum cardinality Die set with respect to s_j can be solved by a transformation to the problem of computing minimum weight edge cutsets in undirected graphs. The relevant definitions are given below.

Definition 3. *Let $G(V, E)$ be an undirected graph with a nonnegative weight $w(e)$ for each edge $e \in E$. Let s and t be two distinct vertices in V. An s-t* ***edge cutset*** *for G is a subset $E' \subseteq E$ such that in the graph $G'(V, E - E')$, there is no path between s and t. A* ***minimum weight*** *s-t* ***edge cutset*** *for G is an edge cutset whose total weight is minimum.*

The following well known result shows that minimum weight edge cutsets can be found efficiently (see for example [28]).

Theorem 1. *Given an undirected graph $G(V, E)$ with a nonnegative weight $w(e)$ for each edge $e \in E$ and two distinct vertices s and t in V, a minimum weight s-t edge cutset for G can be computed in $O(|E| + |V| \log |V|)$ time.* $\square$

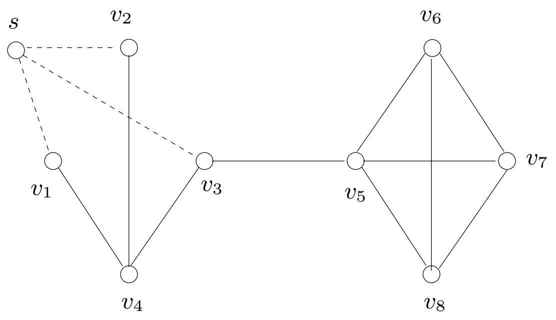
We now explain how an algorithm for the minimum weight s-t edge cutset problem can be used to solve the problem of computing a minimum cardinality Die set for a given service s_j. We need the following definition.

Definition 4. *Let $G(V, E)$ be the given service-oriented network and let s_j be a given service. Let $P_j \subseteq V$ denote the set of nodes that provide service s_j. The* ***auxiliary graph*** *$G_j(V_j, E_j)$ for s_j is an undirected edge weighted graph constructed as follows.*

(a) $V_j = V \cup \{s\}$, where s is a new node that does not appear in V.
(b) $E_j = E_j^1 \cup E_j^2$, where
  (i) $E_j^1 = E - \{\{x, y\} : x \text{ and } y \text{ are both in } P_j\}$ and
  (ii) $E_j^2 = \{\{s, y\} : y \in P_j\}$.
(c) The weight of each edge in E_j^1 is 1 and that of each edge in E_j^2 is ∞.

As an example, the auxiliary graph of the network of Figure 1 with respect to service s_1 is shown in Figure 2. The usefulness of the auxiliary graph is shown in the following lemma.

Lemma 1. *Let $G(V, E)$ be the given service-oriented network and let s_j be a given service. Suppose $G_j(V_j, E_j)$ denotes the auxiliary graph for s_j and v is a node of G that needs service s_j.*



Note: Nodes v_1, v_2 and v_3 provide service s_1. Each dotted edge has weight $= \infty$; other edges have a weight of 1.

Fig. 2. Auxiliary Graph with respect to Service s_1 of the Network in Figure 1

(a) If there is a set $E_v \subseteq E$ such that there is no path in $G'(V, E - E_v)$ between v and any node that provides service s_j, then G_j has an s-v edge cutset of weight at most $|E_v|$.

(b) For any finite integer α, if G_j has an s-v edge cutset of weight α, then there is a set $E_v \subseteq E$ such that $|E_v| = \alpha$, and there is no path in $G'(V, E - E_v)$ between v and any node that provides service s_j.

Proof:
Part (a): Suppose E_v is a set of edges such that there is no path in the graph $G'(V, E - E_v)$ between v and any node that provides service s_j. Let $E_v^1 \subseteq E_v$ be the set of edges obtained by deleting from E_v each edge $\{x, y\}$ such that both x and y are nodes that provide service s_j. Note that each edge in E_v^1 is also an edge in G_j and the total weight of the edges in E_v^1 is at most $|E_v|$. Using the fact that there is no path in $G'(V, E - E_v)$ between v and any node that provides service s_j, it can be verified that E_v^1 is an s-v edge cutset for G_j.
Part (b): For some finite α, suppose $Q_v \subseteq E_j$ is an s-v edge cutset with weight α for G_j. Since α is finite, Q_v cannot contain any edge incident on node s. Thus, each edge in Q_v is also in $G(V, E)$. Using the fact that Q_v is an s-v edge cutset for G_j, it can be verified that there is no path in $G'(V, E - Q_v)$ between v and any node that provides service s_j. This completes the proof. □

The following lemma is a direct consequence of Lemma 1.

Lemma 2. *Let $G(V, E)$ be the given service-oriented network and let s_j be a given service. Let $G_j(V_j, E_j)$ denote the auxiliary graph for s_j. For any node $v \in V$, the minimum number of edges to be deleted from G so that there is no path between v and any node that provides service s_j is equal to the weight of a minimum weight s-v edge cutset in G_j.* □

Input: A service-oriented network $G(V, E)$, the set S of all services, the sets $A(v)$ and $N(v)$ for each node $v \in V$.

Requirement: Find the edge resilience of G.

Algorithm:

```
1. for each service $s_j \in S$ do
      (a) Construct auxiliary graph $G_j(V_j, E_j)$ for service $s_j$.
      (b) Find the set $D_j \subseteq V_j$ of the demand points for service $s_j$ (i.e., the set of
          nodes that need service $s_j$).
      (c) for each node $v \in D_j$ do
             Compute $\alpha_{v,j}$, the minimum weight of an $s$-$v$ edge cutset in $G_j$.
      (d) Let $\sigma_j = \min\{\alpha_{v,j} : v \in D_j\}$.
2. Edge resilience of $G = \min\{\sigma_j : s_j \in S\} - 1$.
```

Fig. 3. Algorithm for Computing Edge Resilience

From Lemma 2, it follows that the cardinality of a minimum Die set with respect to a service s_j can be obtained by computing the minimum weight edge cutset in the auxiliary graph G_j for each pair s-v, where v is a node that needs service s_j. Once we find the cardinality of a minimum Die set with respect to each service s_j, the edge resilience of the given network G can be found by taking the minimum over all services (Observation 1). These observations lead to the algorithm shown in Figure 3 for computing the edge resilience of a given service-oriented network.

We now estimate the running time of the algorithm. Suppose the given network has n nodes, m edges and a total of p services. The running time of the algorithm in Figure 3 is dominated by the minimum weight edge cutset computations. For each service s_j, the algorithm uses $O(n)$ cutset computations. So, the total number of such computations is $O(pn)$. Since each cutset computation can be done in $O(m + n \log n)$ time (Theorem 1), the running time of the algorithm is $O(pn(m + n \log n))$. The following theorem summarizes the above discussion.

Theorem 2. *Given a service-oriented network $G(V, E)$ and the set of service S, the edge resilience of the network can be computed in $O(pn(m + n \log n))$ time, where $n = |V|$, $m = |E|$ and $p = |S|$.* □

3.2 An Algorithm for Computing Node Resilience

Our algorithm for computing the node resilience of a service-oriented network follows the same approach as that of edge resilience. The main difference is that we need to work with node cutsets instead of edge cutsets.

When a subset X of nodes is deleted from a graph $G(V, E)$, each edge incident on a node in X is also deleted. Keeping this in mind, it is straightforward to modify the definition of a deficiency inducing edge (Die) set to obtain the definition of a deficiency inducing node (Din) set.

Definition 5. *Let $G(V,E)$ denote the underlying graph of a service-oriented network. Let S denote the set of all services available in the network. For any subset of nodes X, let $G_X(V-X,E_X)$ denote the subgraph of G obtained by deleting the nodes in X.*

(a) Given a service $s_j \in S$, a set of nodes $X \subseteq V$ is called a ***deficiency inducing node*** *(*DIN*)* ***set*** *for G* ***with respect to service*** *s_j if at least one of the connected components of the graph $G_X(V-X,E_X)$ is deficient with respect to s_j.*

(b) A set of nodes $X \subseteq E$ is a ***deficiency inducing node set*** *for G if at least one of the connected components of the graph $G_X(V-X,E_X)$ is deficient.*

As in the case of edge resilience, it can be seen that a minimum cardinality DIN set for G can be computed by considering each service separately. To compute that value, we use a transformation to the problem of computing minimum weight node cutsets in graphs. The following definition is the node cutset analog of Definition 3.

Definition 6. *Let $G(V,E)$ be an undirected graph with a nonnegative weight $w(v)$ for each node $v \in V$. Let s and t be two distinct vertices in V such that $\{s,t\} \notin E$. An s-t* ***node cutset*** *for G is a subset $V' \subseteq V - \{s,t\}$ such that when the nodes in V' are deleted from G, there is no path between s and t. A* ***minimum weight*** *s-t* ***node cutset*** *for G is a node cutset whose total weight is minimum.*

As indicated by the following result from [10], minimum weight node cutsets can be found efficiently.

Theorem 3. *Given an undirected graph $G(V,E)$ with a nonnegative weight $w(v)$ for each node $v \in V$ and two distinct vertices s and t in V such that $\{s,t\} \notin E$, a minimum weight s-t node cutset for G can be computed in $O(|E|\,|V|^{1/2})$ time.* □

The definition of the auxiliary graph used for computing node resilience is the same as that given in Definition 4, except that edge weights are not used, and the weight of each node is 1. The usefulness of the auxiliary graph is indicated in the following lemma, whose proof is analogous to that of Lemma 2.

Lemma 3. *Let $G(V,E)$ be the given service-oriented network and let s_j be a given service. Let $G_j(V_j,E_j)$ denote the auxiliary graph (with node weights) for s_j. For any node $v \in V$, the minimum number of nodes to be deleted from G so that there is no path between v and any node that provides service s_j is equal to the weight of a minimum weight s-v node cutset in G_j.* □

The rest of the computation is similar to that of edge resilience. The resulting algorithm for computing node resilience is shown in Figure 4. It can be verified that the running time of the algorithm is $O(pm\,n^{3/2})$). The following theorem summarizes the above discussion.

Theorem 4. *Given a service-oriented network $G(V,E)$ and the set of service S, the node resilience of the network can be computed in $O(pm\,n^{3/2})$ time, where $n = |V|$, $m = |E|$ and $p = |S|$.* □

Input: A service-oriented network $G(V, E)$, the set S of all services, the sets $A(v)$ and $N(v)$ for each node $v \in V$.
Requirement: Find the node resilience of G.

Algorithm:

```
1. for each service $s_j \in S$ do
     (a) Construct auxiliary graph (with node weights instead of edge weights) $G_j(V_j, E_j)$ for service $s_j$.
     (b) Compute $D_j \subseteq V_j$, the set of demand points for service $s_j$.
     (c) for each node $v \in D_j$ do
           Compute $\gamma_{v,j}$, the minimum weight of an $s$-$v$ node cutset in $G_j$.
     (d) Let $\Gamma_j = \min\{\gamma_{v,j} : v \in D_j\}$.
2. Node resilience of $G = \min\{\Gamma_j : s_j \in S\} - 1$.
```

Fig. 4. Algorithm for Computing Node Resilience

4 Algorithms for Designing Resilient Networks

4.1 Preliminary Definitions

As mentioned in Section 2.2, the design problem for 1-node or 1-edge resilience assumes that we are given the underlying graph $G(V, E)$, and for each node $v_i \in V$, the set $N(v_i)$ of services needed at v_i. In addition, a nonnegative cost matrix $C = [c_{ij}]$, where c_{ij} is the cost of placing service s_j at node v_i is also given. The goal is to select a set of services to be placed at each node so that the resulting network has the required level of edge or node resilience, and the total cost of placing the services is minimized.

We saw in Section 3 that the analysis problems for edge and node resilience can be solved by considering each service separately. This idea extends to the design problems as well since the placement of one service has no impact on the placement of other services. So, our approach for solving the design problems also considers one service at a time.

Consider any service s_j. Recall that each node v_i such that $s_j \in N(v_i)$ is a **demand point** for s_j. Each node at which service s_j is placed is called a **service point**. A set of service points for s_j is called a **placement** for s_j. Given a connected graph $G(V, E)$ and a placement P for service s_j, we say that the placement is **1-edge-resilient with respect to service** s_j if for every edge $e \in E$, the graph $G'(V, E - \{e\})$ contains a path from each demand point for s_j to a service point for s_j. The definition of a 1-node-resilient placement can be given in a similar manner. Thus, an equivalent way of posing the design problems is the following: find a placement for each service so that the resulting service-oriented network is 1-edge-resilient (or 1-node-resilient) and the total cost of placement is minimized. This formulation is used in the remainder of this paper.

The next two subsections consider the design problem for 1-edge resilience and 1-node resilience respectively. For reasons of space, we will assume uniform cost values for services in this version; that is, we assume that $c_{ij} = 1$ for all i and j. Our algorithms can be extended to nonuniform cost values. These extensions will be included in a longer version of this paper.

4.2 Designing a 1-Edge-Resilient Network

In this section, we develop our algorithm for the design problem for 1-edge resilience. As mentioned above, each service can be considered separately in solving this problem. So, we will focus our attention on one service, say s_j. We say that a placement P for service s_j is **optimal** if P provides 1-edge-resilience with respect to s_j, and $|P|$ is the smallest among all placements which have the resilience property.

To develop our algorithm for this problem, we recall a standard definition from graph theory [29].

Definition 7. *Let $G(V, E)$ be a connected undirected graph. A* ***bridge*** *of G is an edge $\{x, y\}$ whose removal disconnects G. If G has no bridges, then G is called a* ***bridgeless graph****.*

The following is a well known result in graph theory [29].

Lemma 4. *Let $G(V, E)$ be a connected undirected graph. Suppose G has b bridges given by $E' = \{e_1, e_2, \ldots, e_b\}$. Then, the graph $G'(V, E - E')$ has exactly $b + 1$ connected components and each of these components is a bridgeless graph.* □

The bridgeless components (BLCs) of the underlying graph $G(V, E)$ play an important role in solving the design problem. To show this connection, we define another auxiliary graph for G as follows.

Definition 8. *Let $G(V, E)$ be a connected undirected graph and let $E' = \{e_1, e_2, \ldots, e_b\}$ denote the set of bridges of G. The* ***BLC graph*** *of G, denoted by $G_B(V_B, E_B)$, is defined as follows.*

(a) Each node of V_B corresponds to a BLC of G.
(b) For nodes x and y in V_B, the edge $\{x, y\}$ is in E_B if and only if there is a bridge in G that joins a node in the BLC corresponding to x to a node in the BLC corresponding to y.

Intuitively, the BLC graph of G is constructed from G by collapsing each BLC of G into a single super node; the edges of the BLC graph are in one-to-one correspondence with the bridges of G.

Example: The service-oriented network of Figure 1 has one bridge, namely the edge $\{v_3, v_5\}$. One bridgeless component of G is formed by the node set $\{v_1, v_2, v_3, v_4\}$ and the other is formed by $\{v_5, v_6, v_7, v_8\}$. The BLC graph corresponding to this network has two nodes joined by a single edge.

The following is an easy observation concerning the BLC graph of a connected graph G.

Observation 2. *Suppose G is a connected undirected graph. The BLC graph of G is a tree.* □

We need some additional definitions to point out the role played by the BLC graph in solving the 1-edge-resilient design problem. It should be noted that the following definitions are all with respect to the service s_j under consideration.

Definition 9. *Let $G(V, E)$ be a connected graph and let G_B denote the BLC graph of G. Let s_j be a service.*

(a) A ***demand component of*** *G is a BLC of G that has at least one demand point for s_j.*
(b) The ***demand subgraph*** *G_D^j of G consists of nodes v_H corresponding to the demand components of G, and all edges and nodes of G_B that lie along some path connecting two such nodes.*

Since the BLC graph G_B is a tree (Observation 2), the demand subgraph G_D^j is a subtree of G_B. We call each leaf of G_D^j (i.e., a node of degree 1 in G_D^j) a **demand leaf**. Each BLC of G corresponding to a leaf of G_D^j is called a **demand leaf component**. The importance of demand leaf components of G in finding an optimal placements is shown in the following lemma whose proof is omitted because of space limitations.

Lemma 5. *Let G be a connected graph. Let G_B denote the BLC graph of G, and let G_D^j denote the demand subgraph of G_B for service s_j.*

(a) Let δ denote the number of demand leaves of G_D^j. If P^* denote an optimal placement for s_j, then $|P^*| \geq \delta$.
(b) Let P be a placement for s_j obtained by choosing an arbitrary node from each demand leaf component of G. Then, P is 1-edge-resilient. □

Lemma 5 points out that an optimal placement for service s_j can be found by choosing an arbitrary node from each demand leaf component of G. When this process is repeated for each service, we obtain an optimal placement that achieves 1-edge-resilience. The steps of the resulting algorithm are shown in Figure 5.

We now analyze the running time of the algorithm in Figure 5. As in Section 3, let $|V| = n$, $|E| = m$ and $|S| = p$. Finding the BLCs of G can be done in $O(m+n)$ time [3]. Then, constructing the BLC graph G_B can be done in $O(n)$ time, since G_B is a tree. Using a bit vector representation of length p for the set $N(v_i)$ for each node v_i, it can be seen that time used to find the placement for all services is $O(np)$. Therefore, the overall running time of the algorithm is $O(m+np)$. The following theorem summarizes the main result of this section.

Theorem 5. *Given a connected graph $G(V, E)$, the set of services S and the set $N(v_i)$ for each $v_i \in V$, the design problem for 1-edge-resilience under uniform service costs can be solved in $O(m + np)$ time, where $n = |V|$, $m = |E|$ and $p = |S|$.* □

Input: A connected graph $G(V, E)$, the set S of all services, the set $N(v_i)$ for each node $v_i \in V$.

Requirement: For each service s_j, find a service point set P_j (i.e., the subset of V at which service s_j will be placed) so that the resulting network is 1-edge-resilient and $|P_j|$ is the smallest among all placements that provide 1-edge-resilience.

Algorithm:

```
1. Find the bridgeless components (BLCs) of $G$ and construct the BLC graph $G_B$.
2. for each service $s_j \in S$ do
      (a) Compute the demand point set $D_j$ for $s_j$.
      (b) Initialize service point set $P_j$ to $\emptyset$.
      (c) Construct the demand subgraph $G_D^j$ and find its leaf nodes.
      (d) for each leaf $v$ of $G_D^j$ do
             Choose an arbitrary node $w$ from the BLC of $G$ corresponding to $v$, and
             add $w$ to $P_j$.
3. Output the sets $P_j$, $1 \le j \le |S|$.
```

Fig. 5. Algorithm for Designing a 1-Edge-Resilient Network

4.3 Designing a 1-Node-Resilient Network

We now address the problem of computing an optimal placement for achieving 1-node-resilience. The approach is similar to that of the design problem for 1-edge-resilience. We start with some standard graph theoretic definitions [29].

Definition 10. *Let $G(V, E)$ be a connected undirected graph.*

*(a) A node $v \in V$ is a **cut point** (or **articulation point**) if the removal of v disconnects G.*
*(b) A **block** is maximal subgraph G' of G such that G' does not have a cut point.*

Example: Consider the graph $G(V, E)$ shown in Figure 1. It has two cut points, namely nodes v_3 and v_5. G has three blocks: the subgraph induced on the set $\{v_1, v_2, v_3, v_4\}$, the edge $\{v_3, v_5\}$ and the subgraph induced on the set $\{v_5, v_6, v_7, v_8\}$.

Definition 11. *Let $G(V, E)$ be a connected undirected graph. The **block-cut point graph (BC graph)** of G, denoted by $G_B(V_B, E_B)$, is the bipartite graph defined as follows.*

(a) V_B has one node corresponding to each block and one node corresponding to each cut point of G.
(b) Each edge $\{x, y\}$ in E_B joins a block node x to a cut point y if the block corresponding to x contains the cut point node corresponding to y.

The following is a known result about blocks and block-cut point graphs [29].

Lemma 6. *Let $G(V, E)$ be a connected undirected graph.*

(a) Each pair of blocks of G share at most one node, and that node is a cutpoint.
(b) The BC graph of G is a tree in which each leaf node corresponds to a block of G. □

As before, we focus on obtaining an optimal placement for one service s_j. The role of the BC graph of G in the node resilience design problem is similar to that of BLC graph in the edge resilience design problem. To explain this, we need a few more definitions. A node v of G is an **interior demand point**, if v is a demand point (for service s_j) and v is not a cut point. Note that each interior demand point appears in only one block of G.

Definition 12. *Let $G(V, E)$ be a connected undirected graph and let G_B denote the BC graph of G. Consider a service s_j.*

*(a) The **demand subgraph** of G with respect to service s_j, denoted by G_D^j, is the subgraph of G_B consisting of nodes that correspond to blocks of G containing an interior demand point, cut nodes that are also demand points and all edges and nodes of G_B that lie along some path connecting two such nodes.*
*(b) The **pruned demand subgraph** of G with respect to service s_j, denoted by G_{PD}^j, is the subgraph of G_D^j, constructed as follows. If G_D^j consists of a single node, then G_{PD}^j is identical to G_D^j. Otherwise, G_{PD}^j is constructed by removing from G_D^j each leaf node that is a cut point.*
*(c) A demand point v of G is a **generalized interior demand point** if v does not have a corresponding cut point node in G_{PD}^j.*

Since the BC graph G_B of G is a tree (Lemma 6), G_D^j and G_{PD}^j are subtrees of G_B. The following lemma shows the relationship between an optimal 1-node-resilient placement for G and the pruned demand graph G_{PD}^j.

Lemma 7. *Let $G(V, E)$ be a connected undirected graph and let G_B denote the BC graph of G. Let G_{PD}^j denote the pruned demand subgraph of G with respect to service s_j. Let P^* be an optimal placement for s_j.*

(1) Suppose G_{PD}^j consists of a single node.
  (a) If G has only one demand point for s_j, then $|P^*| = 1$.
  (b) If G has two or more demand points for s_j, then $|P^*| = 2$.
(2) Suppose G_{PD}^j consists of two or more nodes. Let δ denote the number of leaves of G_{PD}^j. Then, $|P^*| = \delta$.

Proof sketch: Below, we will indicate how a placement P is constructed. The proof that the placement is optimal and that it provides 1-node-resilience is omitted in this version due to lack of space.

Suppose G_{PD}^j consists of a single node. There are two possibilities here, as indicated in the statement of the lemma. If G has only one demand point v for

Input: A connected graph $G(V, E)$, the set S of all services, the set $N(v_i)$ for each node $v_i \in V$.

Requirement: For each service s_j, find a service point set P_j so that the resulting network is 1-node-resilient and $|P_j|$ is the smallest among all placements that provide 1-node-resilience.

Algorithm:

```
1. Find the cut points and blocks of $G$ and construct the BC graph $G_B$.
2. for each service $s_j \in S$ do
      (a) Compute the demand point set $D_j$ for $s_j$.
      (b) Construct the pruned demand subgraph $G^j_{PD}$.
      (c) Construct the service point set $P_j$ by considering the following cases.
          Case 1: $G^j_{PD}$ has only one node.
             If $G$ has only one demand point $v$ for $s_j$, then let $P_j = \{v\}$. If $G$ has
               two or more demand points for $s_j$, choose any two nodes $x$ and $y$ of
               $G$, and let let $P_j = \{x, y\}$.
          Case 2: $G^j_{PD}$ has two or more nodes.
             (i) Find the leaf nodes of $G^j_{PD}$ and the corresponding leaf blocks of $G$.
             (ii) for each leaf block $H$ of $G$ do
                     Choose an arbitrary generalized interior demand point $x$ of $H$
                     and add $x$ to $P_j$.
3. Output the sets $P_j$, $1 \le j \le |S|$.
```

Fig. 6. Algorithm for Designing a 1-Node-Resilient Network

service s_j, then P consists of just the node v. If G has two or more demand points, then P consists of two arbitrary nodes from G.

Suppose G^j_{PD} consists of two or more nodes. In this case, we identify the blocks of G corresponding to the leaves of G^j_{PD}. Placement P is obtained by choosing from each such block H, an arbitrary generalized interior demand point. □

An algorithm for finding an optimal placement for 1-node-resilience can be constructed from the proof sketch given for Lemma 7. The steps of the resulting algorithm are shown in Figure 6. To estimate the running time of the algorithm, let $n = |V|$, $m = |E|$ and let $p = |S|$. The blocks and cut points of a connected graph $G(V, E)$ can be found in $O(|V| + |E|)$ time [3]. Thus, the BC graph G_B of G can be constructed in $O(n + m)$ time. Consider any service s_j. Using the fact that the pruned demand subgraph G^j_{PD} is a tree, it can be seen that an optimal placement for each service can be found in $O(n)$ time. Hence, the time over all services is $O(pn)$. Therefore, the overall running time of the algorithm is $O(pn + m)$. The following theorem summarizes the above discussion.

Theorem 6. *Given a connected graph $G(V, E)$, the set of services S and the set $N(v_i)$ for each $v_i \in V$, the design problem for 1-node-resilience under uniform service costs can be solved in $O(m + np)$ time, where $n = |V|$, $m = |E|$ and $p = |S|$.* □

5 Summary and Concluding Remarks

We identified some resilience metrics for service-oriented networks. These metrics take into account both the underlying topology of the network and the manner in which services are distributed over the network. We presented polynomial algorithms for determining the edge and node resilience of a given network. We also presented efficient algorithms for optimally distributing services over a given network so that the resulting service-oriented network achieves 1-edge-resilience or 1-node-resilience.

We close by pointing out some directions for future research. First, it would be useful to study the analysis and design problems under Byzantine node and edge failures (instead of passive crash failures). Second, it is of interest to investigate design problems when the placement of one service has an impact on the placement of other services. Finally, it is also important to study other versions of design problems (e.g. adding links of minimum cost to enhance edge or node resilience) to understand the tradeoffs involved in the design of resilient service-oriented networks.

Acknowledgments. We thank the EDCC-5 reviewers for their helpful suggestions.

References

1. G. Brightwell, G. Oriolo and F. Shepherd, “Reserving Resilient Capacity in a Network”, *SIAM J. Discrete Mathematics*, Vol. 14, No. 4, Oct. 2001, pp. 524–539.
2. C. Colbourn, “Network Resilience”, *SIAM J. Algebraic and Discrete Methods*, Vol. 8, 1987, pp. 404–409.
3. T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to Algorithms*, MIT Press and McGraw-Hill, Cambridge, MA, 2001.
4. F. Cuenca-Acuna, R. Martin and T. Nguyen, “Autonomous Replication for High Availability in Unstructured P2P Systems”, *Proc. 22nd IEEE Symposium on Reliable Distributed Systems* (SRDS’03), Florence, Italy, Aug. 2003.
5. S. Czerwinski, B. Zhao, T. Hodes, A. Joseph and R. Katz, “An Architecture for a Secure Service Discovery Service”, *Proc. ACM Symposium on Mobile Computing and Communications* (MobiCom’99), New York, NY, Aug. 1999, pp. 24–35.
6. C. Dabrowski, K. Mills and J. Elder, “Understanding Consistency Maintenance in Service Discovery Architectures in Response to Message Loss”, *Proc. 4th International Workshop on Active Middleware Services* (WAMS’02), July 2002, pp. 51–60.
7. C. Dabrowski, K. Mills and J. Elder, “Understanding Consistency Maintenance in Service Discovery Architectures During Communication Failure”, *Proc. 3rd International Workshop on Software Performance* (WOSP’02), July 2002, pp. 168–178.
8. C. Dabrowski, K. Mills and A. Rukhin, “Performance of Service-Discovery Architectures in Response to Node Failures”, *Proc. 2003 International Conference on Software Engineering Research and Practice* (SERP’03), June 2003, pp. 95–101.
9. J. Douceur and R. Wattenhofer, “Optimizing File Availability in a Secure Serverless Distributed File System”, *Proc. 20th IEEE Symposium on Reliable Distributed Systems* (SRDS’01), New Orleans, LA, Oct. 2001, pp. 4–13.

10. S. Even, *Graph Algorithms*, Computer Science Press, Rockville, MD, 1979.
11. B. Gedik and L. Liu, "Reliable Peer-to-Peer Information Monitoring Through Replication", *Proc. 22nd IEEE Symposium on Reliable Distributed Systems* (SRDS'03), Florence, Italy, Aug. 2003.
12. P. Georgatsos and Y. Joens (Editors), "Towards Resilient Networks and Services", ACTS Guidelines NIG-G5, June 1999.
13. S. Goel, S. Belardo and L. Iwan, "A Resilient Network that can Operate Under Duress: Supporting Communication Between Government Agencies During Crisis Situations", *Proc. Hawaii Intl. Conference on System Sciences*, Jan. 2003.
14. S. Goel, S. Talya and M. Sobolewski, "Service-Based P2P Overlay Network for Collaborative Problem Solving", submitted for publication, March 2003.
15. F. Harary and J. Hayes, "Edge Fault Tolerance in Graphs", *Networks*, Vol. 23, No. 2, March 1993, pp. 135–142.
16. B. Helvik, "Dependability Issues in Smart Networks", *Proc. 5th IFIP Conference on Intelligence in Networks*, Thailand, Nov. 1999, pp. 53–76.
17. F. Hwang, "Comments on 'Network Resilience: A Measure of Network Fault Tolerance'", *IEEE Trans. Computers*, Vol. 43, No. 12, Dec. 1994, pp. 1451–1452.
18. A. Iamnitchi and I. Foster, "On Fully Decentralized Resource Discovery in Grid Environments", *Proc. Intl. Workshop on Grid Computing*, Denver, CO, Nov. 2001.
19. P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1994.
20. S. Jha, J. Wing, R. Linger and T. Longstaff, "Survivability Analysis of Network Specifications", *Proc. 2000 IEEE International Conference on Dependable Systems and Networks* (DSN'00), Workshop on Dependability Despite Malicious Faults, New York, NY, June 2000.
21. D. Loguinov, A. Kumar, V. Rai and S. Ganesh, "Graph-Theoretic Analysis of Structured Peer-to-Peer Systems: Routing Distances and Fault Resilience", *Proc. ACM SIGCOMM'03*, Karlsruhe, Germany, Aug. 2003, pp. 395–406.
22. L. Massoulie, A. Kermarrec and A. Ganesh, "Network Awareness and Failure Resilience in Self-Organizing Overlay Networks", *Proc. 22nd IEEE Symposium on Reliable Distributed Systems* (SRDS'03), Florence, Italy, Aug. 2003.
23. S. Moitra, E. Oki and N. Yamanaka, "Some New Survivability Measures for Network Analysis and Design", *IEICE Trans. Communications*, Vol. E80-B, No. 4, Apr. 1997, pp. 625–631.
24. W. Najjar and J. Gaudiot, "Network Resilience: A Measure of Network Fault Tolerance", *IEEE Trans. Computers*, Vol. 39, No. 2, Feb. 1990, pp. 174–181.
25. D. Pradhan (Editor), *Fault-Tolerant Computing: Theory and Techniques*, Volumes I and II, Prentice-Hall, Englewood Cliffs, NJ, 1986.
26. U. Saif and J. Paluska, "Service-Oriented Network Sockets", Technical Report, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2003.
27. M. Singhal, "Research in High-Confidence Distributed Information Systems", *Proc. 20th IEEE Symposium on Reliable Distributed Systems* (SRDS'01), New Orleans, LA, Oct. 2001, pp. 76–77.
28. M. Stoer and F. Wagner, "A Simple Min-Cut Algorithm", *J. ACM*, Vol. 44, No. 4, July 1997, pp. 585–591.
29. D. West, *Introduction to Graph Theory*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1996.

Efficient Protection of Many-to-One Communications

Miklós Molnár, Alexandre Guitton, Bernard Cousin, and Raymond Marie

Irisa, Campus de Beaulieu, 35 042 Rennes Cedex, France
{miklos.molnar, alexandre.guitton, bernard.cousin, raymond.marie}@irisa.fr

Abstract. The dependability of a network is its ability to cope with failures, *i.e.*, to maintain established connections even in case of failures. IP routing protocols (such as OSPF and RIP) do not fit the dependability objectives of today applications. Usual forwarding techniques based on destination address (like IP) induce many-to-one connections. Thus, if a dependable connection is needed, all primary paths and protections having the same destination must be established in a coordinated way. Otherwise, loops may be established. In this paper, we propose a fault recovery for many-to-one connections based on a cold (preplanned) protection. The main advantage of our approach is that the recovery in case of failures is achieved within a short delay. Additionally, with respect to other approaches, the dependability of the routing scheme is increased in the way that it statistically copes with many failures. The algorithm we propose computes an efficient backup for an arbitrary primary tree using an improved multi-tree algorithm.

Keywords: network, fault-tolerant routing, many-to-one, cold protection, multi-tree algorithm.

1 Introduction

High-speed networks are becoming increasingly important and allows the development of applications with real-time constraints, such as multi-media services, cooperative systems, distributed computing. These applications often rely on the survivability of the network: communications should not be interrupted for a long time by a failure of a link or of a router. Indeed, the longer the communication is interrupted, the more packets are dropped. The problem of fast recovery has been well studied for several types of communications, including broadcast (one-to-all), unicast (one-to-one) and multicast (one-to-many). However, there is no efficient proposition for dependable incast (many-to-one) communications.

Incast connections are many-to-one, , several sources send data to a single destination. An incast connection can be the support of homogeneous or heterogeneous communications. Examples of applications inducing homogeneous communications include log collection, data gathering in sensor networks, auction sales and massive submissions. Data gathering, routing and congestion control problems of many-to-one communications have been analyzed in meshed network for parallel computation cases (. [1, 2]) and in sensor networks (. [3]). To the best of our knowledge, there is no analysis for the dependability and the efficient protection of many-to-one communications. In networks where the forwarding of packets is based on their destination address (such as IP networks), all the communications toward the same destination form an incast connection. In this case, the connection is heterogeneous since it is composed of communications having different requirements and protocols. For example, a FTP communication from a host A to a host C and a HTTP communication from a host B to the same host C form an incast connection. Incast connections are traditionally realized using a tree[1] (sometimes, the tree is implicit, like in IP).

Implementing dependable communications is a major thread for current networks. Indeed, the network is supposed to be survivable, , it should withstand failures of links or routers. Two measures of the dependability of a network can be considered: the recovery delay and the number of failures managed [4]. It is therefore critical to reduce the recovery delay as much as possible. Classical recovery delays of IP protocols such as OSPF or RIP reach tens of seconds (see [5] for OSPF and [6] for the slow convergence problem of RIP). The other measure, the number of failures managed by a recovery mechanism, impacts on the reliability of the network. In our model, we consider two types of failures: independent failures and highly correlated failures. Our proposition supplies a recovery mechanism that protects efficiently against both types of failures.

Our objective is to recover quickly from failures on an incast communication while coping with as much failures as possible. In this paper, we propose a cold preplanned protection that allows local recovery using arc-disjoint trees. Our proposition is a general framework: it can be applied to various networks and layers, , IP networks or switched networks.

Section 2 gives a state of the art on dependable communications. Section 3 describes our protection construction based on arc-disjoint trees. Section 4 describes our local recovery mechanism. Section 5 analyzes the capability of protecting independent failures and highly correlated failures. Finally, we conclude our work in Section 6.

2 State of the Art of Dependable Connections

Several ways to cope with failures exist. In this section, we briefly survey the existing approaches to realize dependable connections while focusing on our main concern: the fastness of the recovery. A detailed survey on survivability can be found in [7] (in the case of WDM networks).

[1] Multicast connections are also realized using a tree, but incast connections and multicast connections are not symmetric: multicast connections and multicast routing protocols require a particular mechanism in routers, the duplication mechanism, while incast connections do not require any.

Hot and Cold Redundancy. Hot redundancy, denoted 1+1 redundancy, consists in sending each message on two disjoint paths simultaneously ([8] for an example). Hot redundancy allows fast recovery since the destination receives the packet from one of the paths even if a failure on the other path has occurred and has not yet been detected. However, hot redundancy wastes a lot of bandwidth. This major drawback and the necessity of a selecting algorithm at the destination make hot redundancy not very used.

Cold redundancy, denoted 1:1 redundancy, consists in raising a recovery mechanism once a failure is detected ([8] again for an example). Although 1:1 redundancies are slower than 1+1 redundancies because of the failure detection delay, they are often preferred since they save bandwidth. The restoration and protection are the two main types of cold redundancy.

Restoration and Protection. Restoration is a reactive approach to cope with failures. At the time a failure is detected, the router that detected the failure searches for a new path to reroute the traffic to the destination [9]. The advantage of restoration is that it adapts to the current state of the network. This kind of solutions can be applied both in the case of highly correlated and multiple independent failures. However, intensive computations are required for the router to find a new path, which increases the recovery delay. Usual Internet routing protocols use this approach.

Protection is a proactive approach to cope with failures. The behavior of the routers in case of a failure is preplanned [10]. At the time the failure is detected, the router reroutes the traffic to the preplanned protection path. This approach has the advantage of being very fast, since the recovery is raised without any additional computation of the router [11]. However, less failures can be managed compared to restoration since protection is proactive. Classic protections are end-to-end or local.

End-to-End and Local. End-to-end recovery consists in rerouting the traffic at the source on an arc-disjoint path, once a failure is detected. A typical example of end-to-end protection is the path-based protection. The delay induced by the end-to-end recovery is high because the source has first to be informed that a failure occurred on the primary path before raising the recovery. Another drawback of path-based protection is that it cannot cope with two successive failures: if a failure occurs on the primary path and another occurs on the alternate path, the connection is interrupted.

Local recovery consists in rerouting the traffic at the router that detected a failure. A typical example of local recovery is the link-based protection. The delay induced by the local recovery is low because the recovery is raised locally. Using link-based recovery, several successive failures on the primary path can be managed, as long as they concern different links. A drawback of link-based protection (but not of local recovery approach) is that node failures are not managed. A comparison of link-based protection and path-based protection can be found in [12].

Typically, local preplanned recovery mechanisms provide protection for multiple independent failures but do not resist highly correlated failures. End-to-end solution can supply a protection in the cases of highly correlated failures if the alternative path is not involved by the failure.

In this paper, we propose a new local cold protection to recover from failures on incast connections within a short delay, and without the drawback of the link-based protection. The specificity of incast connections is the large number of sources; therefore, traditional end-to-end recovery mechanisms are not suited to incast, where all the sources have to be informed of the failures that occurred.

3 Proposed Protection of Incast Trees

Our proposition aims at the construction of dependable incast connections. The dependability of our routing scheme is reinforced since the proposed protection statistically withstands many failures.

Many-to-one connections require the establishment of the primary paths in a coordinated way. The backup paths should be synchronized together and also with the primary paths. We call [illegible] the union of primary paths.

For basic incast connections, the primary tree is usually a shortest path tree. However, QoS aware incast connections may use different partial spanning trees depending on the network management policy. For this reason, we assume that the primary incast tree is given to our algorithm either by the application or by the network management. Often, this primary tree spans only a sub-graph of the given network. The objectives of the protection are: (i) it should work on any topology and (ii) it should protect any given (partial or not) primary tree. Let us denote T_p this given primary tree.

In this section we present how our protection solution can achieve these objectives. Our solution, based on a directed 2-connected structure, builds the backup of a given incast primary tree. We decided to use multi-tree as the directed 2-connected structure. Two questions remain. (i) How to build a multi-tree given a primary tree? (ii) How to build a backup from a multi-tree?

Sections 3.1 and 3.2 describe the construction of a multi-tree. Section 3.3 answers the first question and Section 3.3 answers the second.

3.1 Basic Multi-tree Construction

A [illegible] is a set of two directed trees that are arc-disjoints and that share the same root. The algorithm presented in [13] describes a way to compute such two trees. It works only in undirected, edge-redundant topologies, [illegible], networks that do not have articulation edges. By definition, the deletion of an [illegible] increases the number of connected components. The two directed trees spans all the nodes of the network. We assume that all links are bidirectional.

The multi-tree is built by adding successively external paths, as specified by Algorithm 1. An [illegible] is a path starting in a spanned node u, ending in a spanned node v containing at least one intermediate node and such that all intermediate nodes are out of the multi-tree. An external path is shown on Figure 1. Generally, $u \neq v$ (except at the first iteration or in the case of an articulation node). From each external path, two arc-disjoint branches are extracted such that one of them ends at u while the other ends at v. Each branch starts at the neighbor of u (or v) on the external path and is connected to one of the directed trees. The directions and the connections of these directed branches are determined in a way to ensure the consistency of the trees ([13]).

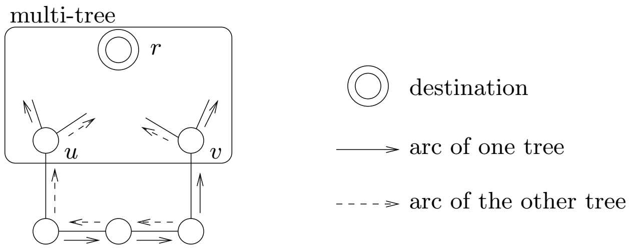


Fig. 1. An external path from u to v

In a node-redundant graph (without articulation node), a multi-tree contains two disjoint paths from any node to the root. Therefore, in the case of a single failure, any node is still connected to the root in the multi-tree.

Algorithm 1 Multi-tree construction

```
Require: a destination node $r$, an edge-redundant graph
Ensure: a multi-tree $M$
  initialize the multi-tree $M$ with node $r$
  while there is an external path of $M$ do
    select an external path $p$
    extract two arc-disjoint directed paths from $p$
    add the two directed paths to $M$
  end while
```

Figure 2 shows the successive steps of the algorithm. In this example, the external path is chosen arbitrarily at each step. The corresponding two arc-disjoint branches are shown on the figure. One of the tree of the multi-tree is represented in solid lines (and is referred to as the blue tree in [13]) while the other is represented in dashed lines (and is referred to as the red tree in [13]).

Advantages. With the help of the multi-tree, tree-based communications can be protected against node and link failures in node-redundant or edge-redundant graphs respectively. The multi-tree is relatively easy to compute and it can be applied to realize hot-redundancy for broadcast or incast communications or to realized cold-redundancy with a preplanned protection.

Drawbacks. One of the drawbacks of the described multi-tree protection is that the algorithm does not deal with arbitrary topologies, only with edge-redundant topologies.

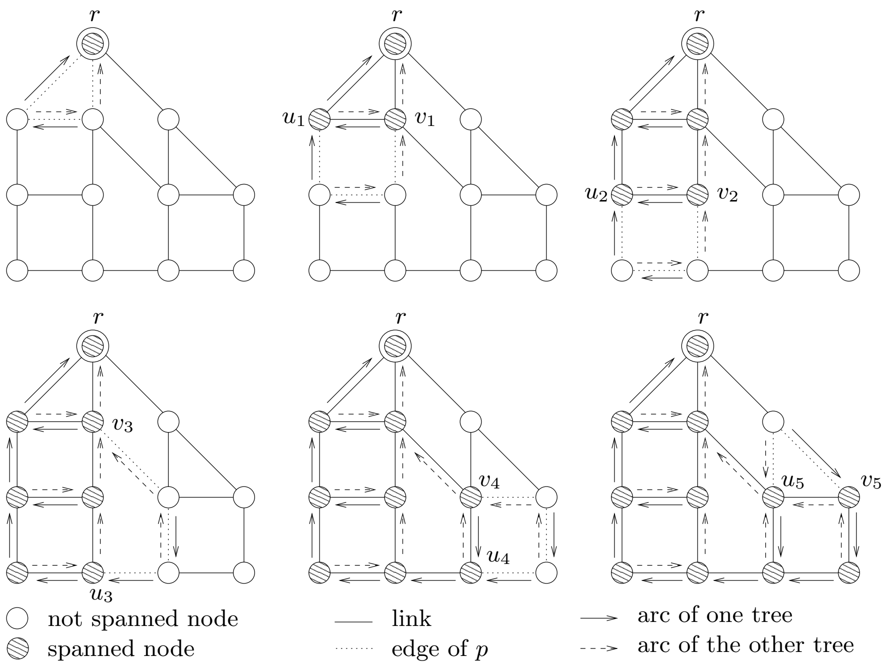


Fig. 2. The multi-tree construction

A more important drawback follows from the fact that the selection of successive external paths is not determined. Thus, the diameter of the trees (and as a consequence, the length of the primary paths) can be arbitrary large (for example, it can be seen on Figure 2 that the distance in the solid tree from node v_5 to the destination r is very long). The QoS requirement of the applications or the network management often impose the use of particular primary trees (, shortest path or QoS aware trees).

In most incast communications, only a sub-set of nodes belongs to the set of the sources and so the incast tree is a partial spanning tree. The algorithm should be adapted to partial spanning trees.

Last but not least, [13] does not present an effective protection mechanism. Our protection mechanism is presented in Section 4.

In the following, we will specify how the different drawbacks of the multi-tree based protection can be eliminated.

3.2 Extension to Arbitrary Topologies

Often, in real networks, articulation edges exist. Even in initially redundant networks, articulations can be produced due to persistent failures. The protection construction should work also in these cases.

To build a multi-tree in an arbitrary connected network, we propose to add to the previous algorithm a particular case when articulation edges are found. In this case, we propose the creation of two directed arcs on the articulation edge toward the destination. So, the two trees of the multi-tree are not arc-disjoints anymore but it tolerates failures on the redundant edges.

Algorithm 2 gives a formal version of this algorithm. The most important improvements are explained deeply in Section 3.3.

Algorithm 2 Special multi-tree construction

```
Require: a destination node $r$, an arbitrary graph
Ensure: a multi-tree $M$
  initialize the multi-tree $M$ with the destination node $r$
  while there remains unspanned nodes in the component connected of $r$ do
    while an external path exists do
      select the optimal external path $p$ according to Subsection 3.3
      extract two arc-disjoint directed paths from $p$
      add these two paths to $M$
    end while
    if an articulation node $a$ is detected then
      select an external path from $a$ to $a$
    else
      if an articulation edge $e$ is detected then
        add the arc of $e$ directed to $r$ to each tree of $M$
      end if
    end if
  end while
```

3.3 Protection of a Total Incast Spanning Tree

In a first time, let us suppose here that all nodes of the network participate to the incast communication. The case of partial participation is discussed in Section 3.4.

Remember that T_p denotes the given primary tree used to transmit data if no failures occur. In networks, failures are rare events. Therefore, the primary tree has to ensure an efficient delivery of data packets and to satisfy certain QoS criterion. Generally, none of the directed trees built by the multi-tree algorithm corresponds to a good primary tree. However, it will be possible to extract from the multi-tree the primary tree T_p (, a shortest path tree) and to use the remains of the multi-tree as the backup of T_p. We will see that this backup is a forest.

To adapt the protection to a given (total) primary tree (for example to the shortest-path tree), we propose the construction of a multi-tree spanning all of the arcs of the given primary tree. This is done using a special external path selection.

Special External Path Selection. To ensure that a given primary tree T_p is covered by the multi-tree, we have to ensure that T_p is covered by the union of the external paths selected during the multi-tree construction. In other words, we have to ensure that every arc of T_p is covered by an arc of the multi-tree.

Let us denote by $dist(T_p, r, n)$ the hop distance from the root r of T_p to a node n in T_p. At each iteration, the algorithm selects an external path containing exactly one edge (n_1, n_2) that is not in the primary tree T_p and with at least one node of $\{n_1, n_2\}$ not spanned by the multi-tree. Such an external path exists if the topology is redundant and if the primary tree is total. If there are several candidates, the one minimizing $dist(T_p, r, n_1) + dist(T_p, n_2, r)$ is chosen. Figure 3 shows the selection of an external path, where T_p is a shortest-path tree.

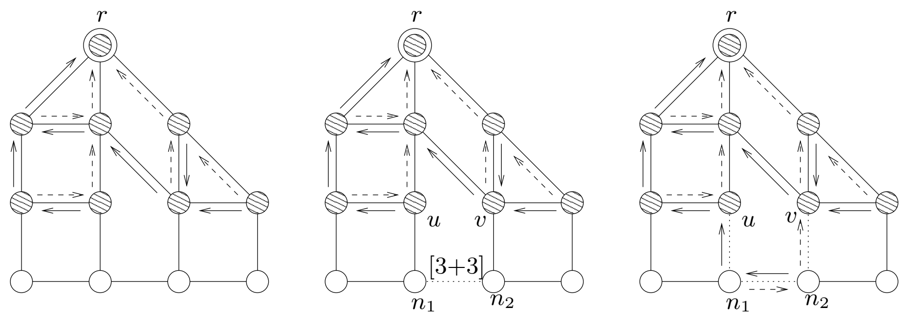


Fig. 3. Selection of the external path $p = \{(u, n_1), (n_1, n_2), (n_2, v)\}$

Having chosen the edge (n_1, n_2), not spanned yet and not on T_p, the external path p can be found as follows: u is the first successor on T_p of n_1 which is covered by the multi-tree, v is the first successor on T_p of n_2 which is covered by the multi-tree. Then, p contains the path on T_p from u to n_1, the edge (n_1, n_2) and the path on T_p from n_2 to v.

Property 1 shows that our selection of external paths ensures that the primary tree T_p is covered by the multi-tree.

The arcs of the directed primary tree T_p are in the multi-tree built by our algorithm.

There is exactly one outgoing arc from each node of the tree T_p to the root r. To prove the property, it is sufficient to show that nodes are added to the multi-tree together with their outgoing arc. Let p be an external path from the node u to v selected at any iteration of the algorithm. In a first time, we show that the outgoing arc of any node $n \in p$ is in p and in a second time we show that there is an arc directed to the same direction in the multi-tree.

1. Let us assume that the outgoing arc of n on T_p is not contained in the path p. In this case, the two adjacent edges of n contained in p are not in T_p. This is impossible by construction, because there is only one edge in p that is not in T_p.
2. The external path maps two outgoing arcs (one in each tree of the multi-tree) to each node, except to u and v. The outgoing arc on T_p of any node n, different from u and v, corresponds to one of the two arcs. □

Minimizing the hop distance $dist(T_p, r, n_1) + dist(T_p, n_2, r)$ allows the protection to be dense. A dense protection is more robust in the case of multiple failures than a sparse protection.

Remember that the outline of the multi-tree construction algorithm with our external path selection is given on Algorithm 2.

Backup Forest Construction. Our improved multi-tree algorithm builds a multi-tree $M = (T_1, T_2)$ that covers the primary tree T_p. To obtain the backup support of the primary tree, Algorithm 3 is proposed.

```
Algorithm 3 Backup forest construction
Require: $T_p$ a primary tree, $M = (T_1, T_2)$ a multi-tree covering $T_p$
Ensure: $F$ is the backup of $T_p$
  $F \leftarrow arcs(T_1) \cup arcs(T_2)$
  $F \leftarrow F \backslash arcs(T_p)$
```

The first step of the algorithm merges the arcs of T_1 and T_2 into a set of directed arcs F. The second step of the algorithm removes in F the arcs of the primary tree T_p. The remaining arcs are the backup protection of T_p. F is a forest. The way the arcs of F are used to protect T_p is described in Section 4.

3.4 Protection of a Partial Incast Spanning Tree

Generally, few nodes participate to an incast communication. Since our preplanned protection scheme is based on a multi-tree, the construction has to take into account the cases where only a sub-set of the nodes are sources of the incast communication. In these cases the resulting preplanned scheme should contain the partial primary tree and an appropriate backup structure, which is compatible with the protection and fault recovery mechanism proposed for total protection cases. For this reason, we propose to build a partial multi-tree which covers the partial primary tree. To create such a partial multi-tree, we propose two algorithms in the next two sections.

Construction Based on the Total Primary Tree. A first way to take partial primary trees into account is (i) to construct the total primary tree for all nodes of the given topology, (ii) to compute the total multi-tree spanning all the nodes and (iii) to prune the parts of the total multi-tree that are not used neither by the partial primary tree nor by its protection. More precisely, an arc of the total multi-tree must be kept if it belongs to the partial primary tree or to a directed path that protects a part of the primary tree, and if this primary tree can not be protected with the help of a shortest path.

Figure 4 illustrates the construction of two partial (and protected) spanning trees based on a total multi-tree. The first figure shows the total multi-tree for the given topology (to simplify the shortest path tree is considered as primary tree).

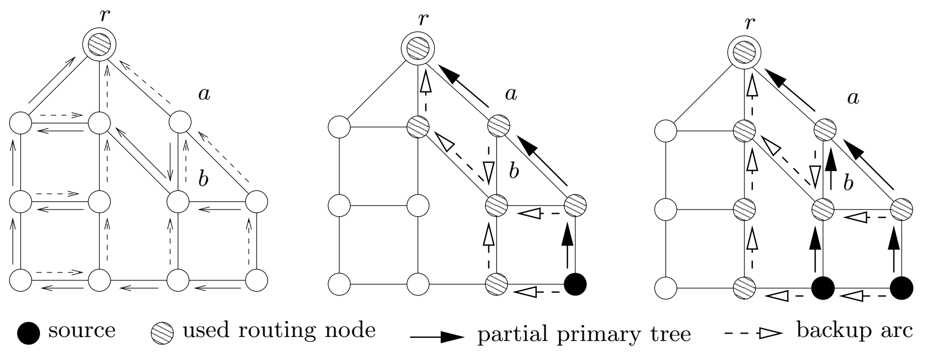
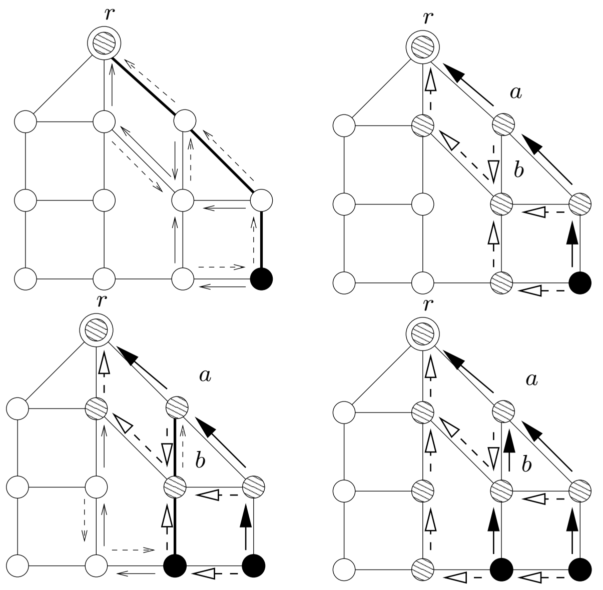


Fig. 4. Partial tree protection resulting from an total multi-tree based scheme

The two following figures present the result of pruning for two different cases. Only the shortest paths and the needed backup arcs are kept. All the arcs not belonging to the primary tree or to its protection must be deleted from the total multi-tree based protection scheme (so the arc (b, a) must be deleted, but the arc (a, b) participating to the protection of (a, r) should be kept in the first case). A new partial primary tree needs the recomputation of the protection scheme. Note that the reconfiguration of the total multi-tree is not necessary; this multi-tree calculated at a first time can be stored and used for ulterior computations. However, this proposition requires an important computation when few nodes are sources of the incast communication.

Partial Construction of the Multi-tree. A more efficient way to take partial primary trees into account is to compute only the required part of the multi-tree and the protected scheme if needed. The most typical case corresponds to the add of a new source to the existing incast protection scheme (the first source being added to an empty scheme). Any partial tree with its protection can be built with successive adds of the sources.

A partial multi-tree covering a primary path can be built by adding successive shortest external paths (loops in the case of the root and articulation nodes) from the destination to the sources by following the primary paths in the reverse direction. If a part of the new primary path belongs to the existing partial primary tree, the construction starts with the last node of the new path belonging to the tree (see Fig. 5). Each external path should cover the next edge of the primary path. The external path selection and the stop condition of the multi-tree algorithm should be modified accordingly. Note that nodes which are in the partial multi-tree but not in the primary tree have two directed paths to the destination. The arcs belonging neither to the primary tree nor to its protection can be deleted. Figure 5 illustrates the successive partial multi-tree construction and the obtained protection scheme for two sources. The left upper part and the left bottom part of the figure shows the partial multi-tree construction for the given source and primary path. The right part of the figure, the obtained protection scheme, shows the established partial primary tree T_p in solid lines and its backup forest F in dashed lines.



—— shortest path from the source to the tree

Fig. 5. Successive constructions of a protected partial incast tree

If the graph is edge-redundant, the proposed backup forest F insures that there are two arc-disjoint paths from all of the router nodes of the primary tree to the destination.

Trivially, one of the directed paths is the primary path. Let us suppose that the topology is edge-redundant. In the case of a multi-tree spanning the entire network graph, there are two arc-disjoint paths from all of the nodes to the destination.

In the case of a partial spanning tree, the primary path is covered by successive external paths (the first path containing the destination is a loop). The partial multi-tree is built by creating two arc-disjoint directed paths on the selected external paths. If there is no articulation edge in the primary path, then in the partial multi-tree there are two arc-disjoint paths from all of the nodes to the destination. Since only the non used arcs (which are not included to the primary tree nor to its simple protection) are deleted, there are always two-arc-disjoint paths from nodes of the primary tree to the destination. □

4 Proposed Recovery Mechanism

This section deals with our recovery mechanism. This mechanism uses the arcs of the backup forest to protect against failures. Once the protection has been configured, a local recovery has to be implemented in routers to protect the incast connection against failures.

Our mechanism is not specific to a network: it can be applied to several networks, including IP and switched networks and is not related to a specific layer. It may be implemented in layers such as Layer 2 (, Ethernet), Layer 3 (, IP) or Layer 7 (application) networks. Our hypotheses are the following:

– the network topology has to be undirected,
– the protection computation and the configuration: (i) are centralized in a manager and (ii) the manager has to know the topology.
– (i) routers have to store an alternative table in addition to their forwarding table and (ii) their forwarding algorithm is slightly modified.

If the router has only one outgoing arc in the protected scheme, then this next hop should be in the primary forwarding table for the given destination. If there is a second (backup) outgoing arc, then this latter should be in the alternative forwarding table of the router. If a router detects a failure on the primary entry of a destination r, it switches its primary entry to its alternative entry. This failure detection algorithm is described by Algorithm 4. Note that the way the failure is detected is out of scope of this paper.

Algorithm 4 Failure detection

```
Require: an interface $i$
Ensure: all primary entries corresponding to $i$ are switched
  if a failure is detected on interface $i$ then
    for all primary entries pointing to $i$ do
      switch on the alternative entry (the primary entry is replaced by the alternative
      entry and the alternative entry is dropped)
    end for
  end if
```

A router n detects the need for a route modification when it receives a packet for a destination r on an interface i which corresponds to the next hop for r on the primary path: a loop appears between n and its next hop. In this case, n switches its primary entry to its alternate entry to avoid the loop. This failure propagation occurs only when the backup path uses the edges of the primary path in the reverse direction. For example, if node a of Figure 5 detects that the link (a, r) failed, it switches to its alternate entry (failure detection). Then, the next packets toward r reaching a are forwarded to b. When receiving a packet from a for r, b detects that a is the next hop to r on its primary path. Thus, b switches to its alternate entry (failure propagation). The next packets will follow the backup path up to r. Algorithm 5 describes the forwarding algorithm of the routers.

Of course, the failure of the destination node or of an articulation node or edge may disconnect the incast connection. In this case, there is obviously no way to maintain the connectivity. However, our recovery mechanism ensures that there is no infinite loop of the traffic in any connected component of the network. Figure 6 shows that loop can occur during the recovery, but their duration is

Algorithm 5 Forwarding and failure propagation

```
Require: a packet $p$ for a destination $r$ is received on an interface $i$
Ensure: $p$ is sent to the right interface
  if the primary entry for $r$ is equal to $i$ then
    switch on the alternative entry (the primary entry is replaced by the alternative
    entry and the alternative entry is dropped)
  end if
  if the primary entry exists then
    send $p$ according to the primary entry
  end if
```

very short: the packets are forwarded by a node at most twice before a stable state is reached. On Figure 6, the configuration (A) is before the failure, (B) is during the (limited) unstable state and (C) is the configuration obtained at the stable state.

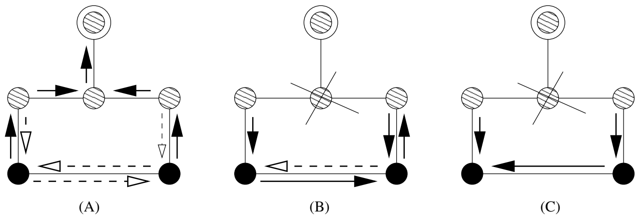


Fig. 6. A failure may disconnect the network, but the traffic will not enter into an infinite loop

5 Analysis of the Dependability in the Case of Multiple Failures

In this section, we analyze the dependability of several protections in case of multiple failures. The number of failures that can be managed by a protection greatly depends on the network topology and on the incast connection. To study the dependability of a protection in the case of multiple failures, we propose to discuss on the number of failures that do not interrupt communications. In the following our proposition is compared to the two well-known cold protection schemes: the path-based and the link-based protections.

5.1 Independent Failures

In our model of independent failures, we assume that failures occur successively on the primary path. Indeed, failures that do not occur on a primary path do not impact communications.

Path-Based Protection. When the first failure occurs on the primary path and the source is informed, the traffic is swapped to the backup path at the source. Then, if a second independent failure touches the backup path, the communication is interrupted.

Link-Based Protection. Link-based protections do not cope with node failures. In the case of a link failure, the bypass is used to reach the next node while the rest of the primary path is used. If an ulterior link failure occurs on the primary path, another bypass can be used if and even if the second bypass is link-disjoint with the first bypass. The link-based protection allows independent link failures if the bypasses are independent. However, it does not cope with failures which occurs on the bypasses.

Our Protection. The proposed incast tree protection copes with a link or a node failure. Similarly to the link-based protection, ulterior failures on the primary path can be recovered. If a second failure touches the backup used to recover from a first failure, there are two possibilities. If this backup corresponds to a primary path of another communication and its backup is different from the first primary path, then the local recovery is possible. If the failed part of the backup does not correspond to a primary path of another communication or if the backup of this failed point uses the first failed primary path, then the mechanism cannot cope with the failure. The mechanism is illustrated on Figure 7 (A) and (B) in the case of link and node failures.

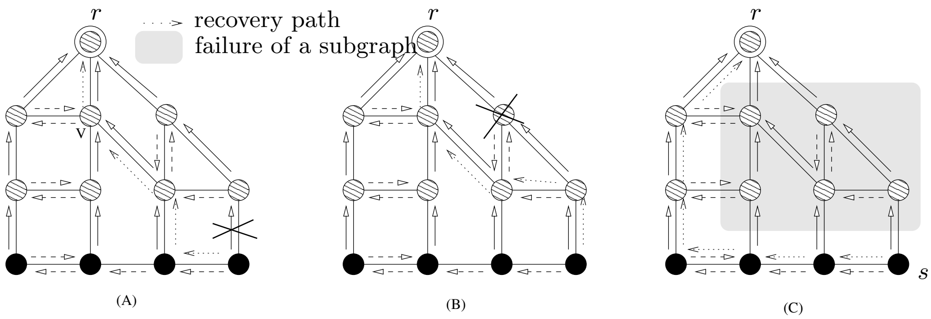


Fig. 7. Recovery capability of our mechanism

5.2 Highly Correlated Failures

In our model of highly correlated failures, the failures occur simultaneously on adjacent links or nodes of the network: a connected subgraph of the network fails. The higher the protection resists to highly correlated failures, the higher can be the diameter of the failed subgraph without interrupting communications. In the same way than previously, we assume that at least a link or a node of primary tree fails.

Path-Based Protection. Simultaneous failures can occur on the links and nodes of a primary path. The path based protection copes with large diameter of failures when all internal nodes and edges of the primary path failed. However, the alternate path should be intact.

Link-Based Protection. Since link-based protection does not cope with node failures, the maximal diameter of a recovered failure is one link.

Our Protection. In redundant topologies, the proposed tree-based protection gives a link-disjoint alternate path from all the nodes of the primary path to the destination. Highly correlated failures on a primary path with large diameter can be recovered. Moreover in the case of dense incast communications, the mechanism copes with failures which occur on the alternate paths. Figure 7 (C) shows the maximal sub-set of the network which can fail with recovery from a source s.

6 Conclusion

In this paper, we presented a cold protection for many-to-one communications. The protection computation is based on the construction of a backup forest, given an arbitrary primary tree. The recovery uses a simple failure propagation mechanism and can be realized by a local switch operation in the concerned routers. Thus, it produces low failure recovery delay. We studied the impact on the protection of two scenarios of failures: independent failures and highly correlated failures. We were able to show that our method can manage efficiently independent failures and highly correlated failures. In the future, we intend to evaluate quantitatively our protection through simulations.

References

1. Krizanc, D., Zhang, L.: Many-to-one packet routing via matching. In: Annual International Conference on Computing and Combinatorics, Shanghai, China (1997)
2. Makedon, F., Simvonis, A.: Many-to-one packet routing on the mesh. In: Annual Conference on Computer Science. (1999)
3. Ee, C.T., Bajcsy, R.: Congestion control and fairness for many-to-one routing in sensor networks. In: International Conference on Embedded networked sensor systems, Baltimore, USA (2004)
4. Duato, J.: A theory of fault-tolerant routing in wormhole networks. IEEE Transactions on Parallel and Distributed Systems **8** (1997) 790–802
5. Goyal, M., Ramakrishnan, K.K., Feng, W.C.: Achieving Faster Failure Detection in OSPF Networks. In: IEEE ICC. (2003)
6. Obradovic, D.: Formal Analysis of Convergence of Routing Protocols. PhD thesis, University of Pennsylvania (2001)
7. Maier, G., Pattavina, A., De Patre, S., Martinelli, M.: Optical Network Survivability: Protection Techniques in the WDM Layer. Photonic Network Communications **3/4** (2002) 251–269

8. Anand, V., Qiao, C.: Static versus dynamic establishment paths in WDM networks, Part I. In: IEEE ICC. Volume 1. (2000) 198–204
9. Ramamurthy, S., Mukherjee, B.: Survivable WDM Mesh Networks, Part II – Restoration. In: IEEE ICC. Volume 3. (1999) 2023–2030
10. Ramamurthy, S., Mukherjee, B.: Survivable WDM Mesh Networks, Part I – Protection. In: IEEE INFOCOM. Volume 2. (1999) 744–751
11. Nakamura, R., Ono, H., Nishikawara, K.: Reliable switching services. In: IEEE Globecom. (1994)
12. Johnson, D., Johnson, G.N., Beggs, S.L., Botahm, C., Hawker, I., Chng, R.S.K., Sinclair, M.C., O'Mahony, M.J.: Distributed restoration strategies in telecommunications networks. In: IEEE ICC. Volume 1. (1994) 483–488
13. Médard, M., Finn, S.G., Barry, R.A.: Redundant trees for preplanned recovery in arbitrary vertex-redundant or edge-redundant graphs. In: IEEE/ACM Transactions on Networking. Volume 7. (1999) 641–652

Impact of Faults in Combinational Logic of Commercial Microcontrollers[*]

D. Gil, J. Gracia, J.C. Baraza, and P.J. Gil

Fault Tolerant Systems Group (GSTF),
Department of Computer Engineering (DISCA),
Polytechnic University of Valencia, Camino de Vera s/n,
46022 Valencia, Spain
{dgil, jgracia, jcbaraza, pgil}@disca.upv.es

Abstract. This work shows that faults affecting the combinational logic embedded in a microcontroller can propagate to register elements and may have an important impact over applications, even in the most favourable case of short transient faults. Using VHDL-based fault injection techniques, we have experienced that the percentage of propagated faults, and thus their influence in the microcontroller upper layers, increases as clock frequencies rise. Experiments confirm that single faults can corrupt a number of registers at a time, this number being greater as the duration of the fault increases. From the application viewpoint, results show that, in some cases, faults can lead applications to fail in more than 80% of the cases, which suggests the need of improving the error detection and recovery mechanisms of existing commercial microcontrollers.

1 Introduction

The ever decreasing price/performance ratio of microcontrollers makes them today a suitable choice for assuming the role traditionally played in many embedded systems by other mechanical or electronic components. This motivates an increasing importance of studying the dependability of embedded systems. An approach for characterising the behaviour of microprocessors and microcontrollers in presence of faults relies on the consideration of hardware faults affecting registers and memory elements [1], [2], [3]. Nevertheless, as well as VLSI ICs integration density and frequencies rise, so the incidence of faults in combinational logic (due to causes such as cosmic radiation, electromagnetic interference, transients in power supply, etc.) increases [4], [5].

The goal of this work has been to study of the effects of faults in combinational logic of commercial microcontrollers. To verify the impact of these faults, a broad set of permanent and transient faults has been injected into the models of two real microcontrollers, using a VHDL-based fault injection tool called VFIT [6]. Faults

[*] This work has been partially sponsored by the European Project "Dependability Benchmarking" (DBench), IST-2000-25425, and the Spanish Project "Mejora de las técnicas de inyección de fallos en modelos VHDL", TIC-2002-02491.

have been injected into the ALU and the general clock line of the microcontrollers. The propagation of faults to CPU registers and the failures provoked have been studied (see Fig. 1).

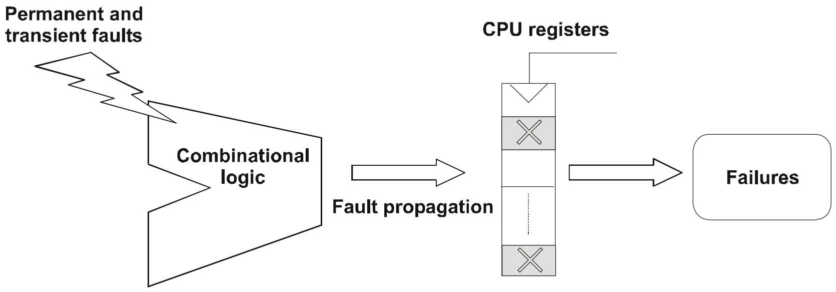


Fig. 1. Propagation of faults from combinational logic to CPU registers, and failures provoked

Fault injection experiments have been carried out with the specific purpose of obtaining answers to the following questions:

- Which percentage of faults do corrupt the CPU registers?
- How many registers are corrupted in average by each *propagated* fault? That's to say, is there a single or a multiple corruption?
- Which is the percentage of failures provoked by injected and propagated faults?
- Which is the influence of the frequency of the system clock in the impact of the faults?

Our results show that, as theoretically pointed by other authors [4], the presence of multiple faults in registers and the influence of the frequency are two key effects related to the impact of faults in deep submicron technologies. The study of fault propagation also gives useful information to common fault injection techniques at register level, such as SWIFI (Software Implemented Fault Injection) and FPGA-based fault injection [7].

The paper is organised as follows. Section 2 describes briefly the target systems. Section 3 discuses the fault models injected. Section 4 depicts the fault injection environment. Section 5 describes the experiments carried out, and Section 6 explains the results obtained. Finally, Section 7 presents some conclusions and future work paths.

2 Target Microcontrollers

We have considered two typical commercial 8-bit microcontrollers: PIC16C5X [8] and MC8051 [9], frequently used in simple embedded computer systems. The former has a RISC architecture with two-stage pipeline and a reduced set of register oriented instructions. The later presents a more conventional Harvard architecture.

In the VHDL models of the microcontrollers, the larger part of the combinational logic belongs to the ALU. So this is the main target chosen to inject the faults. Also, we have injected faults in the general clock line of the microcontrollers, which is a critical point of failure in the system.

Injected faults propagate to the microcontroller registers, and in some cases they provoke the failure of the application.

3 Fault Models

As far as fault models are concerned, our intention has been to use models that would represent real physical faults that occur in ICs. As VLSI ICs integration density and frequencies rises, the most frequently used models (stuck-at (0, 1) for permanent faults, and bit-flip for transient faults) have been observed to be insufficient [10]. Table 1 summarises the fault models selected in this work to be injected with the fault injection tool. This table shows the fault models for combinational logic in CPU. Fault models have been deduced from the physical causes and mechanisms implied in the occurrence of faults, at technological and electronic level [11], [12]. The final selection has been determined by the capabilities of the fault injection technique employed: simulator commands.

Table 1. Fault models selected for fault injection experiments

Transient Faults	Permanent Faults
Delay, Pulse, Indetermination	Stuck-at (0,1), Delay, Open-line, Indetermination

Next we explain briefly the meaning of the fault models used:

- Pulse: provoked by soft errors[1] in combinational logic. It is different from bit-flip because after the fault disappears, the output value returns to its correct value. That is, it behaves as a pulse.
- Delay: The transmission delay of the signal is modified. It is related to transients in power supply and crosstalk.
- Indetermination: Signal value changes to an indeterminate logic value, between ‘0’ and ‘1’. It is related to intermediate voltage values and crosstalk.
- Stuck-at (0,1): Signal value is fixed to ‘0’, ‘1’.
- Open-line: Signal value is high impedance (open[2]).

There are some other fault mechanisms related with permanent faults shown in Table 1: oxide damage (mainly oxide breakdown, hot carrier injection and plasma damage), metallization damage (such as electromigration, stress voiding, contact migration, via migration and microcracks) and package and assembly faults.

[1] When a charged particle hits the combinational logic block, it generates a transient current pulse that provokes a transient voltage pulse in the combinational nodes. This phenomenon is called Single Event Transient (SET) [13].

[2] Short faults, traditionally applied togheter with open faults in fault modelling, cannot be injected with the fault injection technique used (simulator commands). Anyway, intermetallic shorts lead at higher abstraction level to indetermination, stuck-at and delay faults in CMOS technology.

4 Fault Injection Environment

The injection experiments shown in this work have been carried out using VFIT, a VHDL-based fault injection tool developed by our group. In this Section, both the tool and the injection technique are described.

4.1 The Fault Injection Tool

As mentioned, to perform the injection experiments we have used VFIT [6], [14] whose main features are:

- It has been developed to be used in PC platforms or compatible.
- VFIT has been built around a commercial simulator, Modelsim® [15], by Model Technology®.
- It can inject faults automatically applying simulator-commands, saboteurs and mutants techniques.
- It is able to inject permanent, transient and intermittent faults.
- It has a wide set of fault models, surpassing the classical stuck-at (for permanent faults) and bit-flip (for transient faults).
- VFIT can inject faults into the VHDL model of a system and analyse their effects. The analysis can be either the study of the Error Syndrome of the system or the Validation of the system fault tolerance mechanisms.
- It has a complete graphic interface to allow the user to configure the injection campaigns specifying multiple parameters, related to the model, the injection of the faults, and the type of study carried out.

Fig. 2 shows the block diagram of VFIT. The main components of the tool are:

- The *Syntactical and Lexicographical Analyser* generates the *syntactical tree* of the model.
- The *Graphic Interface*, whose mission is to help the user to specify all the parameters needed to perform the injection campaign, and stores the parameters in two files. To show the injection targets, it makes use of the *syntactical tree* of the model.
- The *Injection Manager* performs two important tasks:
  - First, it generates (from the configuration files) the set of parameters required in every injection experiment: fault place, fault model, injection instant, fault duration, etc.
  - Second, using the parameters generated previously, it creates a script file (see Section 4.2) containing the simulator commands needed to perform each particular fault, and sends it to the commercial simulator to be executed. After each simulation, a trace file is generated.
- The *Result Analyser* compares every trace file to a *golden run*, and according to the type of analysis to be carried out, it extracts and calculates a number of parameters and measurements.

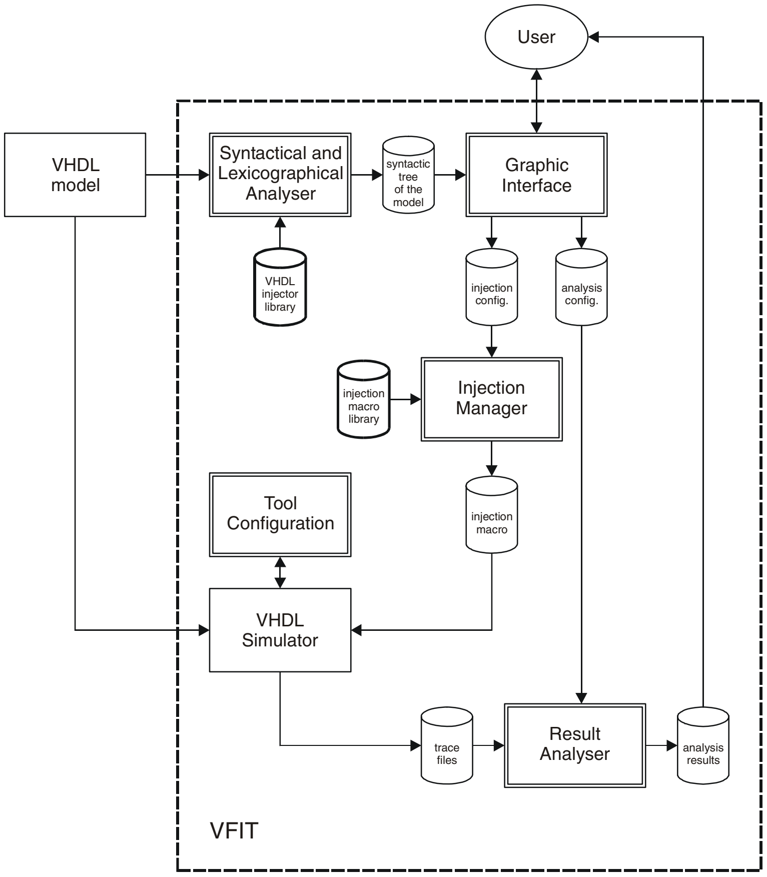


Fig. 2. Block diagram of the fault injection tool

4.2 Fault Injection Using Simulator Commands Technique

This technique is based on using at simulation time the commands of the simulator to modify the value or timing of the signals and variables of the model [16].

The way that faults are injected depends on the injection place. To inject faults on signals, the following sequence of pseudo-commands must be performed:

```
Simulate_Until [injection instant]
Modify_Signal [name] [faulty value]
Simulate_For [fault duration]
Restore_Signal [name]
Simulate_For [observation time]
```

This sequence is thought to inject transient faults, which are the most common and difficult to detect. To inject permanent faults, the sequence is the same, but omitting steps 3 and 4. To inject intermittent faults, the sequence consists in repeating the five steps, with random separation intervals.

The sequence of pseudo-commands to inject faults on variables is:

```
Simulate_Until [injection instant]
Assign_Variable [variable name] [faulty value]
Simulate_For [observation time]
```

The operation is similar to the injection on signals, but there is no control of the fault duration in variables.

Usually, the sequence of real simulator commands can be written in a script file that the simulator can execute.

5 VHDL-Based Fault Injection Experiments

The objective of the work presented in this paper can be framed as the study of Error Syndrome. In the experiments, faults have been injected into the combinational logic of PIC and 8051 microcontrollers, and their propagation to the CPU registers and the failures provoked have been studied.

The general conditions of the different experiments are described below:

- **Fault injection technique:** We have used the simulator commands technique, because it is easy to apply, provides excellent controllability and observability, and its temporal cost is the lowest of all the techniques.
- **Number of injected faults:** 3000 or 5000 single faults per experiment. This gives acceptable values for confidence intervals.
- **Workload:** We have chosen two simple algorithms in order to limit the simulation time. Also, the objective has been to sensitise a big part of CPU registers. These algorithms are Arithmetic Series and Bubblesort.
- **Injection place:** Faults have been randomly injected into any atomic combinational signal of the ALU, as well as into the general clock line of the processor.
- **Fault models:** For each single fault, the fault model has been selected randomly among the models shown in Table 1.
- **Injection instant:** It is generated randomly in the range [0, $t_{workload}$], where $t_{workload}$ is the duration of the simulation of the workload without faults.
- **Simulation duration:** The simulation duration time is the execution time of the workload plus a short period of observation.
- **Fault duration (for transient faults):** It has been generated randomly in the ranges [0.1T-1.0T], [1.0T-10.0T] and [10.0T-20.0T], where T is the CPU clock cycle duration. Also, in some experiments, we have injected transient faults of fixed duration: 10 ns, 100 ns and 1000 ns. It has been intended to inject short-

duration faults, whose duration is equal to a fraction of the clock cycle as well as "long" faults, which will ensure in excess the propagation of faults.

– **Measures:** In order to quantify the study of fault propagation and the provoked failures, we have defined the following measures and calculations to be obtained:
  - Number of corrupted registers ($N_{corrupted}$). A register is considered corrupted if it has at least a faulty bit.
  - Number of propagated faults ($N_{propagated}$). A fault is considered propagated if at least one register is corrupted.
  - Percentage of propagated faults:

$$P_P(\%) = \frac{N_{propagated}}{N_{injected}} \times 100 , \tag{1}$$

  where $N_{injected}$ is the number of injected faults.
  - Number of failures provoked by propagated faults ($N_{propagated\text{-}failure}$). We consider a *failure* as a deviation of the delivered service of the workload from the compliance with the specification.
  - Percentage of failures provoked by propagated faults:

$$P_{PF}(\%) = \frac{N_{propagated-failure}}{N_{propagated}} \times 100 \cdot \tag{2}$$

  - *Multiplicity*, or average number of corrupted registers per propagated fault:

$$Multiplicity = \frac{N_{corrupted}}{N_{propagated}} \cdot \tag{3}$$

6 Results and Discussion

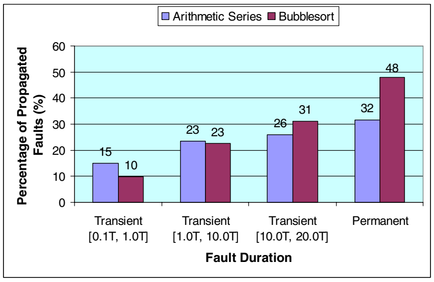
Next we show some significant results derived from the experiments.

6.1 Percentage of Propagated Faults

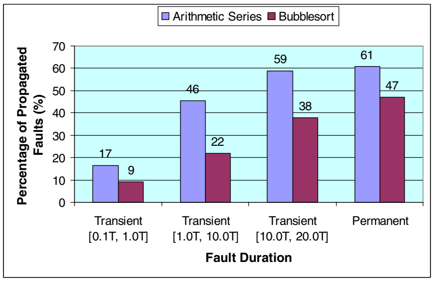
Fig. 3(a) shows the percentage of propagated faults for the PIC (calculated with Eq. 1). It can be observed that the percentage increases with the fault duration. As expected, long-duration faults produce a higher perturbation in the system. Also, not negligible differences between the two workloads are observed, particularly in long-duration faults.

If we consider short-duration faults and Arithmetic Series workload, 15% of injected faults are propagated to CPU registers and provoke the corruption of one or more registers. This means that if we inject 5000 faults, an average of 750 faults propagate to registers. And this occurs in the best case, because the fault impact increases notably as fault duration augments.

Fig. 3(b) repeats the results for the 8051. The general trends are the same, although we can observe higher differences between the two workloads.



(a)



(b)

Fig. 3. Percentage of propagated faults. (a) PIC. (b) 8051

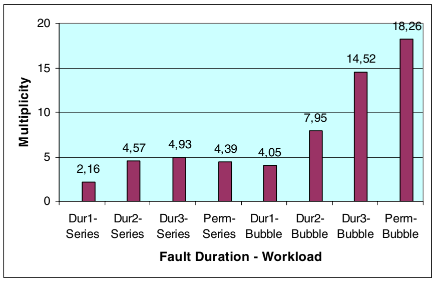
6.2 Corruption of Multiple Registers

To analyse in more detail the sensitivity of CPU registers to faults injected in combinational logic, we have calculated (using Eq. 3) the average number of corrupted registers per propagated fault, for all the previous cases. We have defined this as *multiplicity* (of corrupted registers).

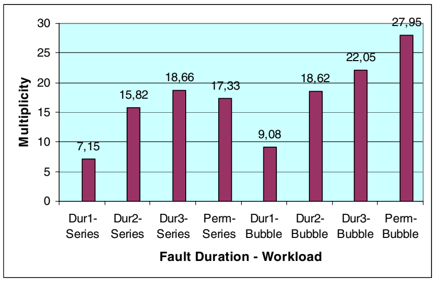
Fig. 4 shows some results related to this matter. Here, *Dur1*, *Dur2* and *Dur3* represent the transient fault duration cases previously considered in Fig. 3.

Fig. 4(a) shows the values of multiplicity for PIC, considering both workloads. Values between 2 and 5 corrupted registers are observed for Arithmetic Series. In Bubblesort case, the multiplicity is quite higher, with values between 4 and 18 corrupted registers. In this case, long-duration faults increase notably the multiplicity. It seems that more complex workloads (like Bubblesort) can lead to higher values of multiplicity, because they sensitise more registers.

Fig. 4(b) repeats the results for the 8051. The general trend is the same, although we can observe even higher values of multiplicity than in the PIC (up to 28 registers!).



(a)



(b)

Fig. 4. Multiplicity of faults in registers. (a) PIC. (b) 8051

In summary, simulations have shown that a single fault in combinational logic can perturb several registers at a time. Moreover, we have found quite high values of multiplicity for the two microcontrollers.

6.3 Provoked Failures

In order to verify the impact of corrupted registers on the application results, the percentage of failures provoked by faults propagated to registers (P_{PF}) has been calculated (see Eq. 2). Fig. 5(a) shows this information for the PIC. It can be seen that the percentage of failures increases with fault duration for both workloads, with values from 32% to 70% in Arithmetic Series case and from 13% to 88% in Bubblesort case.

For instance, in case of short-duration transient faults and Arithmetic Series, 32% of propagated faults cause failures. This corresponds to approximately 5% of injected faults, as can be easily calculated from the expression:

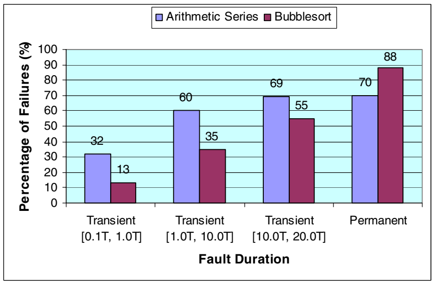
$$\frac{N_{propagated\_failure}}{N_{injected}} = P_P \times P_{PF} \ , \tag{4}$$

taking data from Fig. 3(a) and Fig. 5(a). This means that if we inject 5000 faults, approximately 250 of them will provoke a failure.

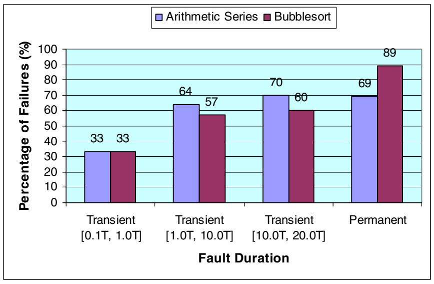
In case of short-duration transient faults and Bubblesort, 13% of propagated faults produce failures. This corresponds to approximately 1.3% of injected faults.

Fig. 5(b) shows the same information for 8051. In this case, the impact of faults is higher. Values from 33% to 69% for Arithmetic Series, and from 33% to 89% for Bubblesort, are observed. In case of short-duration transient faults, 33% of propagated faults produce failures. This corresponds to approximately 6% of injected faults for Arithmetic Series workload and 3% for Bubblesort workload.

We can conclude that the percentage of failures provoked by faults in combinational logic is not negligible, even when having short duration. Moreover, this percentage increases strongly for long-duration faults. This fact suggests the need



(a)



(b)

Fig. 5. Percentage of failures provoked by propagated faults. (a) PIC. (b) 8051

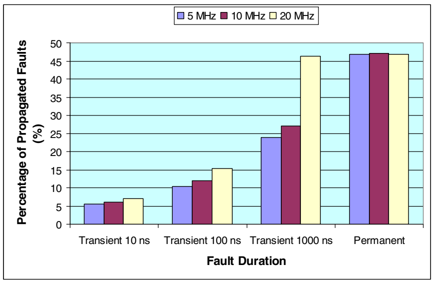
of improving the error detection and recovery mechanisms to tolerate these faults in reliable embedded systems. Some low-cost techniques, in addition to the typical internal exceptions of the processor are: Parity codes for error detection in CPU registers, Hamming codes for error correction in CPU registers and integrated watchdog timers to detect control-flow errors.

6.4 Influence of the Processor Frequency

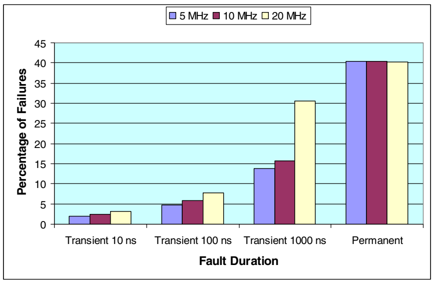
Finally, we have carried out some experiments to study the influence of the clock frequency of the processor. The results come from a representative case: the 8051 microcontroller running Bubblesort. The main conclusions obtained are:

- When injecting transient faults, the increase of the frequency provokes a raise of the percentages of propagated faults in registers. In fact, higher frequencies will increase the probability of storing in flip-flops the transient erroneous data generated at combinational logic.
  Fig. 6(a) shows this situation for different fault durations. Long-duration faults produce higher differences between frequencies. As expected, no dependency on frequency for permanent faults is observed.
- The percentage of failures also increases with the clock frequency, as Fig. 6(b) reflects. In this case, the percentage $\frac{N_{propagated-failure}}{N_{injected}} \times 100$ has been calculated. It is similar to P_{PF}, but normalised to $N_{injected}$ instead of $N_{propagated}$. As it was seen before, long-duration faults produce higher differences between frequencies. As before, no dependency on frequency is observed for permanent faults.

In conclusion, we have verified the influence of the clock frequency on the impact of faults produced in combinational logic. As the clock frequency increases, the impact is also higher. Even, it seems that the dependency is not linear, with bigger raises for higher frequencies. This is very important in VLSI integrated circuits, where frequencies rise continuously.



(a)



(b)

Fig. 6. Influence of the clock frequency (8051 running Bubblesort). (a) In the percentage of propagated faults. (b) In the percentage of failures

7 Conclusions and Future Work

In this work, a study of the impact of faults in commercial microcontrollers has been performed. We have carried out a set of fault injection experiments using VFIT, a general VHDL-based fault injection tool. Faults have been injected into the combinational logic of two real microcontrollers, PIC16C5X and MC8051. The fault propagation to register elements, and the failures provoked, have been studied. It has been also verified the influence of the frequency of the clock system on the impact of faults.

Some significant conclusions extracted from experiments are:

- The percentage of propagated faults is important, even for short-duration faults (between 9% and 17% of injected faults). This is the best case, because the impact of faults increases notably for long-duration faults.
- Simulations have shown that a single fault in combinational logic can corrupt several registers at a time. Moreover, we have found quite high values of multiplicity varying in a wide range according to various factors: the workload, the fault duration and the microcontroller. Roughly, the values are between 2 and 10 in short-duration faults, and between 15 and 30 in long-duration faults. This suggests the need of taking into account multiple faults when injecting faults at register level by means of techniques such as SWIFI.
- The percentage of failures provoked by faults in combinational logic is important. In fact, we have obtained values of up to 33% of propagated faults for short-duration faults. This percentage increases strongly for long-duration faults, reaching up to 80% in some cases.
- As the clock frequency increases, the impact of faults is also higher. Even, it seems that the dependency is not linear, with bigger raises for higher frequencies. This can be important for VLSI integrated circuits, where frequencies rise continuously.

In summary, results show that faults in combinational logic have a notable effect on the system behaviour, even in the most favourable case of short-duration transient faults. This fact suggests the need of improving the error detection and recovery mechanisms to tolerate these faults in reliable embedded systems.

On the other hand, the present work claims for a more detailed study of the effect of multiple faults in registers. In addition, it is also interesting to investigate to what extend our conclusions are applicable to other microcontrollers (such as those with more complex RISC pipelined architectures). Finally, analysing fault models at higher abstraction levels in order to deduce more pertinent faults applicable to behavioural VHDL models, seems necessary as well.

References

1. Carreira, J., Madeira, H., Silva, J.G.: Xception: a Technique for the Experimental Evaluation of Dependability in Modern Computers. IEEE Trans. on Soft. Eng., 24(2):125–136 (1998)
2. Velazco, R. and Rezgui, S.: Transient Bitflip Injection in Microprocessor Embedded Applications. Procs. 6th Int. On-Line Testing Workshop (IOLTW). Palma de Mallorca, Spain (2000) 80–84

3. Gaisler, J.: A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture. Procs. 2002 Int. Conf. on Dependable Systems and Networks (DSN). Washington, D.C., USA (2002) 409–415
4. Constantinescu, C.: Impact of Deep Submicron Technology on Dependability of VLSI Circuits. Procs. Int. Conf. on Dependable Systems and Networks (DSN). Washington, D.C., USA (2002) 205–209
5. Shivakumar, P., Kistler, M., Keckler, S.W., Burger, D., Alvisi, L.: Modeling the Effect of Technology Trends on Soft Error Rate of Combinational Logic. Procs. Int. Conf. on Dependable Systems and Networks (DSN). Washington, D.C., USA (2002) 389–398
6. Baraza, J.C., Gracia, J., Gil, D. Gil, P.J.: A Prototype of a VHDL-Based Fault Injection Tool: Description and Application. J. of Systems Architecture, 47(10):847–867 (2002)
7. Benso, A. and Prinetto, P. (eds.): Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation. Kluwer Academic Publishers (2003)
8. Microchip Technology Inc.: PIC16C5X. Available at: http://www.microchip.com
9. Oregano Systems: MC8051 IP-Core. Available at: http://oregano.at/services/8051.htm
10. Amerasekera, E.A. and Najm, F.N.: Failure Mechanisms in Semiconductor Devices. John Wiley & Sons (1997)
11. Gil, P. *et al.*: Fault Representativeness. Deliverable ETIE2 of Dependability Benchmarking Project (DBench), IST-2000-25245 (2002)
12. Baraza, J.C.: Contribución a la Validación de Sistemas Complejos Tolerantes a Fallos. Nuevos modelos de fallos y técnicas de inyección de fallos. Ph. D. Dissertation. DISCA-UPV (2003)
13. Leavy, J. *et. al.*: Upset due to a single particle caused propagated transient in a bulk CMOS microprocessor. IEEE Trans. on Nuclear Science 38(6)1493–1499 (1991)
14. Gil, D., Baraza, J.C., Gracia, J., Gil, P.J.: VHDL simulation-based fault injection techniques. In [7] 159–176
15. Model Technology: Modelsim SE User's Manual, Version 5.5e (2001)
16. Arlat, J., Boué, J., Crouzet, Y.: Validation-Based Development of Dependable Systems. IEEE Micro 19(4):66–79 (1999)

Implementation of a Testing and Diagnostic Concept for an NPP Reactor Protection System

Tamás Bartha[1], István Varga, Alexandros Soumelidis, and Géza Szabó[2]

[1] Computer and Automation Research Institute,
Hungarian Academy of Sciences,
H-1111 Budapest, Kende u. 13–17, Hungary
{bartha, ivarga, soumelidis}@sztaki.hu

[2] Department of Transport Automation,
Budapest University of Technology and Economics,
H-1111 Budapest, Bertalan Lajos u. 2, Hungary
szabo-g@kaut.kka.bme.hu

Abstract. This paper presents the concept and practical realization of the testing and diagnostic methodology for a reactor protection system in a nuclear power plant. The test concept utilizes the highly redundant nature of these systems to conduct tests during the normal operation of the monitored process. The diagnostic concept uses a simple rule-based expert system to locate the root cause of the failure indications. The diagnostic system can be integrated into a risk monitoring solution, which monitors and computes the functional degradation of the most important safety activities of the protection logic. The described principles have been implemented in the form of microcomputer-based testing and diagnostic systems.

1 Introduction

The instrumentation and control (I&C) systems used in safety-critical application areas are gradually replaced by software-based digital I&C systems. Such refurbishment of the supervision and monitoring systems has recently taken place (or will take place in the near future) in many nuclear power plants (NPPs) of Europe. Despite the obvious advantages, such as the additional functionality provided, the economical and operational benefits, and the pressure driving the adoption of software-based I&C technology in safety-critical environments, there are also important disadvantages/concerns that must be addressed [6]:

- The highly integrated functionality requires a more rigorous specification compared to the partitioned functionality of analog systems;
- The unique properties arising from sampling and quantization of input signals and variable space must be dealt with;
- The increased system complexity and the two-way (HW/SW) development cycle multiplies the potential design and operational faults.

The Reactor Protection System (RPS) of an NPP is a supervisory system that is responsible for the continuous monitoring and the safe shutdown of a reactor unit in an emergency situation. RPSs, as all physical systems, are also susceptible to faults: sensor failures and component hardware faults as well. Naturally, each RPS has a fault-tolerant, highly redundant hardware and software architecture. But the occurring faults may cause latent errors that become activated only in non-nominal or even emergency conditions. Since new RPSs are realized as computerized systems, in theory the built-in hardware self-tests (BIST) could cover almost the whole spectrum of the possible single hardware component failures. Of course, practical constraints such as limited time, performance and QoS requirements limit the real fault coverage.

The remaining failure modes not covered by BIST must be detected by periodically executed tests with human interactions. RPSs must be tested by special test equipment during normal operation to assure reliable operation in all situations (by eliminating latent errors). This testing methodology is called *periodic testing* [3]. Periodic test equipment must be able to create experimental scenarios of input signals in one part of the protection system without affecting the operation of the other parts and the RPS as a whole.

Our paper introduces a periodic and outage testing methodology for reactor protection systems. The practical implementation of this methodology was realized in the form of computerized testing equipment, which is in current use in the Paks Nuclear Power Plant located in Hungary. The Paks Nuclear Power Plant was founded in 1976 and started its operation in 1981. The plant operates four VVER-440/213 type reactor units with a total nominal (electrical) power of 1860 MW. All of the four reactor units have a dedicated Reactor Protection System (RPS), whose purpose is the monitoring and supervision of the attached unit, and especially, the intervention and safe shutdown of the unit in an emergency situation. We describe the test machine hardware and the corresponding test phases.

2 Test Concept

The monitoring and control equipment of potentially dangerous industrial processes, such as the nuclear energy production process, are *safety-critical* systems. Fault tolerance of these devices can and must be realized with many different techniques, all of which incorporates some form of redundancy. In the nuclear industry the primary goal of the fault-tolerant design is to eliminate any *single-point of failure*, while providing single fault identification capability. Therefore, the applied technique in nuclear plants is N-modular (most frequently triple or quadruple) redundancy, combined with design diversity, validity testing, synchrony testing and voting [1].

The nuclear process has different distinct states. Between some of these states the changes are abrupt (nominal operation and emergency situation are good examples). There are physical devices and logical functions that are either not operational or not testable in the normal state, such as the Emergency Protection

Logic (EPL) and Emergency Core Cooling Logic (ECCL). Latent errors in the inactive parts of the system may cause failures in the emergency protection devices at the worst possible moment. Testing must be performed in order to detect such latent errors.

2.1 Testing Triple-Modular (2/3) Systems

The inherent redundancy of the safety-critical supervisory systems offers a good opportunity for on-line testing. Triple- and quadruple-modular systems incorporate three or four realizations of the same protection logic. Therefore, in a fault-free system one of the redundancies can be (logically, not physically) "taken out" of the system (e.g., by invalidating the output signals of the redundancy in order to get it eliminated at the voting stage). Then, the detached redundancy can safely be set into any desired state and tested [10]. In the remainder of the paper we will consider only triple-modular 2/3 systems for two main reasons: 1) they represent the worst-case situation, 2) the Paks NPP Reactor Protection Systems are based on 2/3 redundancy (see Fig. 1 for the structure of the EPL and ECCL logics).

Test Conditions. During the test of a chosen redundancy (further on also referred to as a *train*) from a 2/3 redundancy system, only two trains remain in normal operation. This situation is very sensitive, since an occurring error in the operational part of the system cannot be tolerated, and thus could initiate an emergency protection action. Therefore, there are strict conditions of testing, represented in a test initiation sequence. In the first step, the train to be tested

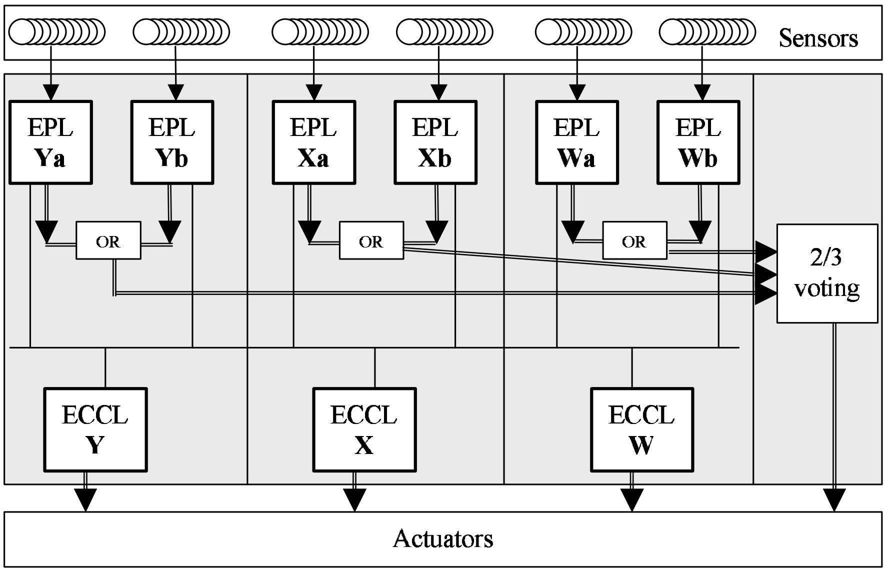


Fig. 1. Safety-critical systems in the Paks NPP

must be manually selected by means of the train selection switch. This activates the `Test Enable` (TE) signal of this train and disables the TE signals of other trains. The TE signal is limited in time by a timer function in order to prevent "forgotten" test select signals. There is also a Test Enable button on the operator panel of the RPS, this must be pressed in the second step by the unit operator on duty to authorize testing. And thirdly, the operational part of the RPS must be error-free before and during testing, any error detection signal cancels test execution [2].

During the test the process parameters are set in a way that safety actions are initiated. The structure and signal processing of the RPS implies that in this situation all ECCS trains would be actuated simultaneously. To guarantee that they can be tested train-by-train, communication interlocks that prevent the tested train of influencing the other ECCS trains have to be established within the RPS logic during the test.

Test Stop Criteria. As mentioned before, the test execution makes the 2/3 redundancy systems sensitive to occurring errors. For this reason, the two trains not tested are constantly monitored for errors during the test. If an error is detected in these systems, the test stop criterion resets the TE signal. Consequently, the execution of the test immediately breaks, and the tested train returns into normal operation. This assures that two fault-free trains are available for correct voting, and thus avoids unnecessary EP actions. The TE signals of not tested trains are disabled in case of appearance of test stop criteria in any of the trains of the RPS. Test stop criteria are ignored only when the `Outage Switch` is set.

2.2 Test Procedure

Two main types of test can be carried out during the normal operation of the RPS system: *built-in self-tests* (these are standard tests provided by the system software) or external *in-service inspection tests* (performed by the test machines described in later sections). In-service inspection tests are necessary to demonstrate that the actuation channels from sensors to the actuators (including wiring and instrumentation) are free of faults. These tests are performed periodically (e.g., on a daily, weekly or monthly basis), depending on the fault behavior of the monitored component or function. Another set of in-service inspection tests is executed during the refuelling of the reactor. They are referred to as 'periodic tests' and 'outage tests', respectively, in contrast with the so-called 'self-tests' performed permanently during the normal operation of the system [4].

The Tested System. The scope of the tests carried out during periodic testing is determined by the development and maintenance procedure of the Reactor Protection System itself. The RPS of the Paks NPP is based on a general-purpose I&C system with safety-critical extensions: the TELEPERM/XS (TXS) system of Siemens. The TXS is essentially a scalable distributed computing system. The architecture is fine-grained, replaceable hardware units are functional modules such as the power supplies, processing modules, communication control processors, analogue and digital input/output boards, etc. The hardware is proven to be single-fault tolerant.

In the software development phase, the process function requirements are analyzed, structured and based on the specification of I&C functions. A graphical engineering tool is used to specify the I&C system functions by means of detailed function diagrams. These diagrams provide the common basis for system documentation and design verification, as well as all the project-specific software. The application software code is derived from these detailed function diagrams by means of automatic code generation tools. Code generation is a strictly rule-based conversion of the functional specification data. The generated code implements the functions in a hardware-independent form, and complies with IEC 60880 requirements. The application software code can be analyzed by a tool and verified against the specification data in the database.

Test Types. For the above mentioned reasons periodic tests aim at detecting failures in the *hardware* of the RPS, thus it is not their purpose to detect design or coding faults in the software. During the periodic tests always the same test procedures are performed. Therefore, any deviation in the response of the RPS is due to an undesired change in the fault state of the hardware. Periodic tests are high-level functional tests of the RPS specified operation. There are four main test types related to the RPS functional module types (see Table 1):

Table 1. Types of RPS periodic tests

Test type	Subtype	Tested unit	Testing method
Neutron fluxes test		NF computers	physical I/O signals
Input module test		I/O modules	physical I/O signals
Functional tests	TS functional test	TS computers	L2 protocol telegrams
	VT functional test	VT computers	L2 protocol telegrams
	VT output test	actuators	L2 protocol telegrams
Startup/Outage tests		ECCS operation	physical I/O signals

Neutron Fluxes Test. During the neutron fluxes tests the test machine sends a start signal to the fluxes generation module. This module gives a programmed variable neutron value to the RPS inputs, while the test machine checks responses from the RPS logic.

Input Module Test. The test machines are able to generate analog and binary signals and drive the RPS inputs with these values. The input module tests check the precision of the analog input modules and the appropriate logic signal comparison level of the digital input modules in the RPS.

Functional Tests. In functional tests a corresponding pair of L2 communication lines that connect the tested train with the other two trains are disconnected, and replaced by the pair of L2 communication lines built-in the test machine. During this procedure the L2 connections of the tested redundancy remain alive. Then, the test machine can modify the state of a well-defined subset of RPS signals (certain input and internal signals as well) by sending L2 telegrams to the tested train via the connected communication lines. The tests are performed in all of the different replacement combinations.

Startup/Outage Tests. During plant start-up, after refuelling, the Emergency Core Cooling System (ECCS) systems are tested. The objective of startup tests is to inject process parameters that initiate safety actions. These tests use the input signals to load the test values into the reactor protection system, similarly to input module tests.

2.3 The Universal Test System (UTS)

The principles shown above have been implemented in the form of microcomputer-based test machines for the Reactor Protection System of the Paks Nuclear Power Plant. Three realizations of the test concept were created: two separate test machines for the startup and operational phase of the reactor, and later (on the basis of field experience) an improved test system, which includes the complete functionality of the earlier machines.

The firstly developed device, called the *Startup Test Machine* (STM) executes short tests on the trains of the RPS before restoring the unit in operation from the outage state, in order to verify the availability of emergency protection actions. The STM is able to generate physical test signals (both current signals in the 0-20 mA range and voltage signals 0-24 V range) and apply them to the inputs of the RPS. The inputs are updated with a 100 ms cycle time. A test script controls the signal generation process, while the so-called Service Unit computer continuously monitors the output signals of the RPS. The STM connects to the Service Unit as a client on the Ethernet network, and obtains the sampled and recorded input and output signals. After the test an automated procedure evaluates the results and declares the test to be passed or failed.

The second device, called the *Periodic Test Machine* (TM), executes the periodic tests. The TM provides a set of mobile computing services to achieve testing and verification of both the input modules and the functionality of the RPS system. TM equipment is divided into two sections related to the components of general-purpose computation and the signal interface function of the unit. The signal interface is a custom design, consisting of special purpose signal conditioning, switching and isolating units. The TM is able to generate physical signals and apply them to the inputs of RPS, much like the STM. However, it is also capable to communicate with the RPS using telegrams conforming the L2 PROFIBUS protocol. The test signals and the modified telegrams change the state of trains.

The original periodic testing concept of the RPS system suggested a standalone Test Machine temporarily connected to several test inputs and communication interfaces of the system to perform test procedures. However, the Periodic Test Machine designed and realized on this basis proved to be inconvenient and difficult to use in practice. The revised concept includes a distributed test system, which has *unified test functionality* to provide more convenient and efficient

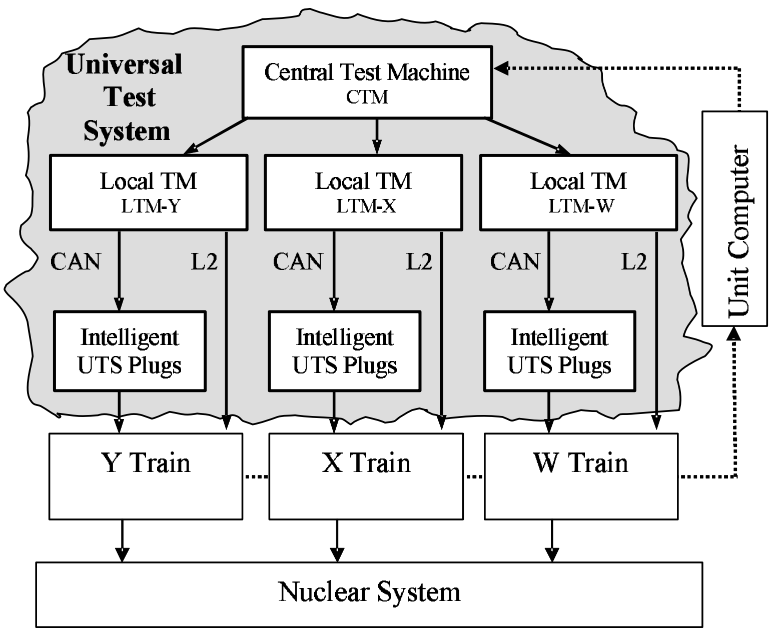


Fig. 2. Universal Test System architecture

means of performing both startup and periodic tests. This new test equipment is called the *Universal Test System* (UTS).

The architecture of the UTS is shown in Fig. 2. The Central Test Machine is supervising the Local Test Machines. This machine is a normal PC with an Ethernet network adapter and the UTS CTM software, which provides the user interface to the system. The Local Test Machines (there are three of them connected to each redundant train of RPS) receive the test sequence from the CTM, and communicate these to the Active Test Plugs. The task specific Test Plugs are microprocessor controlled analog (see Fig. 3) and binary active plugs that contain all the devices that are necessary to perform a full test of the analog and binary input modules of the RPS. The active control plug contains transceiver and receiver components to send and receive control signals to/from the RPS system (position codes, test-release and test enable signals, test trigger, etc.). The basis of the communication between the LTM and a connected test plug is the Control Area Network (CAN), an RS485-type industrial bus system. Design of the active test plugs, the microcomputers and the RS485 bus interface requires realization in high density SMD technology.

The active test plugs are powered by the RPS system when all safety requirements are equally satisfied. Galvanic isolation must only be realized on the RS485 interface. The analog and binary test plugs can be applied simultaneously to multiple input modules, even in a whole cabinet or a complete train. The interconnection between the active plugs and the connection with the host PC (Local Test Machine) is realized via light twisted-pair cables. A significant part

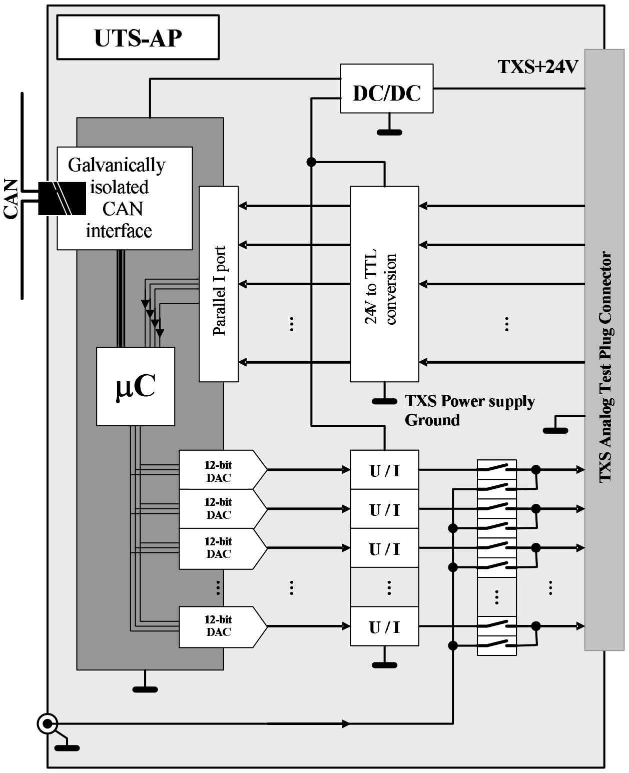


Fig. 3. UTS Analog Intelligent Plug design

of the periodic tests is based upon the L2 interconnection with the RPS. The L2 tests require two independent RS485-based PROFIBUS lines. The L2 tests can be performed with a Local Test Machine by applying either PROFIBUS cards.

3 Status and Risk Monitoring

The Reactor Protection System (RPS) is one of the most vital control systems of Nuclear Power Plants. However, its main functions (the Emergency Core Cooling System and the Emergency Protection System) are only triggered during the periodic testing in normal operation, which is infrequent. Therefore the continuous monitoring of the actual status of the RPS is also essential. This task is traditionally done by the built-in self-test (BIST) circuitry, an obligatory part of every safety-critical hardware [5].

Unfortunately, built-in hardware tests cannot always isolate the real cause of the detected errors. In many cases just the functional degradation can be identified, e.g., a failure of a communication processor can be detected by other communication processors as a communication failure. This is partly due to the complex structure of the RPS system. Another factor is the application indepen-

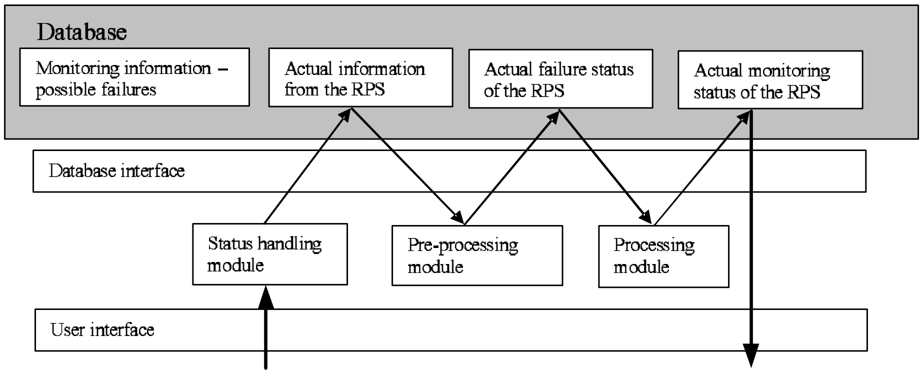

Database
Monitoring information – possible failures
Actual information from the RPS
Actual failure status of the RPS
Actual monitoring status of the RPS
Database interface
Status handling module
Pre-processing module
Processing module
User interface

Fig. 4. Status monitoring system for the RPS

dent nature of the built-in self tests, and therefore the loose correspondence of BIST fault indications with the purpose and structure of the RPS. The consequence is a large amount of failure indications generated on the system, providing good fault *detection* information, but contributing less to fault *localization*.

3.1 Fault Localization

A decision system was developed in order to solve the above mentioned problems as part of the plant information system [12]. The main purpose of this so-called *status monitoring subsystem* is the isolation of the root-cause faults from the indicated failures. The fault indications come from various sources: BIST error signals, physical and logical invalid signal states due to limit violations, the absence of synchrony among trains, etc. However, results of periodic testing are *not* integrated into this process. The inference procedure was implemented as a simple rule-based production system. Its general scheme contains three main parts: the current status of the monitored system, the rule base representing the expert knowledge about the fault manifestation process, and the evaluation engine or processing module (see the scheme shown in Fig. 4). The status of the RPS is obtained from the Unit Computer, which is a large scale plant-wide distributed information processing and archival system.

The status monitoring process is carried out in three main steps. The information coming from the RPS (representing the detected phenomenon) is stored in a central database. This information is treated as the highest level of information (the actual failure indication signals). In the second step, the highest-level information is deduced to an intermediate level, where possible erroneous system states are identified. Finally, the actually faulty hardware devices (and, if identifiable, also their failure modes) are located in the third step. In our current realization, the decision system creates an *a posteriori* evaluation of the archived RPS signals. This functionality is useful for the operators to quickly analyze an already occurred incidence, or for the support personnel to locate the faulty hardware after the reactor has been shut down. Later, we plan to extend the status monitoring subsystem to a continuous implementation, using on-line

```
IF   GW_A_MSI_MCRe_H1_segment_possible_failure = TRUE AND
    (MSI_MCRe_CPU1_(z054)_processor_possible_failure = FALSE OR
     MSI_MCRe_CPU2_(z055)_processor_possible_failure = FALSE)
THEN GW_A_MSI_MCRe_H1_segment_fault = TRUE
```

Fig. 5. Sample rule from the RPS diagnostic rule base

signals from the RPS, and automated triggering of the fault localization and subsequent risk calculation procedures.

The decision system contains the inference rules in textual format, using the well-known "`IF` logic expression `THEN` implication" notation. Thus, the rule base can easily be edited by the end-user without the help of the developer. In the following example, `GW_A_MSI_MCRe_H1_segment_possible_failure`, `MSI-_MCRe_CPU1_processor_possible_failure` and `MSI_MCRe_CPU2_processor-_possible_failure` belong to the set of failure statuses of the Reactor Protection System (Fig. 5). These are binary variables and their value derives from the actual status of the RPS. Since the collected information contains a lot of more complex indicators (such as 2-out-of-3 voter deviations, 2nd max./2nd min. deviations, etc.), some indicators have to be processed to obtain the values of real component failure statuses.

The event `GW_A_MSI_MCRe_H1_segment_fault` is a member of the set called *actual monitoring status* of the system and consequently represents a root cause of a malfunction. The members of this low-level fault set can be displayed directly to the operators or to the maintenance personnel.

3.2 Risk Monitoring

The decision system identifies the root causes of the detected failures. This information can be used for maintenance purposes, but it is furthermore useful for actual risk calculation as well. The actual risk calculation (*risk monitoring*) can help the operators in the control room to separate the events requiring immediate actions from events with a small impact on safety. Risk monitoring is a step towards the usage of a risk-based operational scenario instead of the traditional event-based one [11].

The main source of the risk in an NPP Reactor Protection System are the actuation masking and false actuation events occurring in the previously mentioned two main subsystems: EPS and ECCS. During design verification certain limits are set for the probabilities or frequencies of the root events, and it must be demonstrated that the requirements are met by the designed architecture and by the proposed operational and maintenance policies. For this purpose fault-tree analysis is widely used, because of its sound methodology and the large number of well-tested software tools. The fault trees contain some basic information, such as the reliability model and parameters describing component failures (these are called basic events). By augmenting the fault-trees with actual plant information regarding the status of basic events, fault-tree analysis becomes useful for actual risk estimation as well (in addition to the verification of the design).

For this purpose, up-to-date plant information has to be collected and the parameters of the fault-tree must be modified according to the current situation. In our solution, this information is provided by the status monitoring subsystem. Since the FT models of the RPS were established in the RPS design phase and are continuously maintained, these models can be directly applied for the purpose of risk monitoring [7]. When a given failure occurs, the status monitoring system calculates the actual monitoring status and informs the risk monitoring system about the changes. The risk monitoring system initiates a Risk Spectrum analysis, modifying the fault trees by passing the parameters corresponding to the actual failures to the analysis tool. These parameters set the probability of related gates and/or basic events to occurred status, thus the updated fault-tree will correspond to the actual RPS state.

One of the key practical issues related to the implementation of the risk monitor is the long execution time of the analysis. A possible solution to shorten the execution time is *pre-calculation*. During a given plant state, all of the non-active failure events (basic events) in the fault-tree are set to the triggered state, respectively, and the top event parameters are calculated. These calculated figures are stored in loop-up tables for later reference. Considering that faults are individual and there is a certain time between faults, only the pre-calculated result must be selected when a new fault occurs. The fault-tree analysis tools often support the type of analysis required for the described pre-calculation - this is called the *Risk Increase Factor* (RIF), which can show how the individual failure can affect the whole system.

4 Conclusions

In this paper we presented the concept of operational testing and diagnosis of an important class safety-critical supervisory systems: the Reactor Protection System of nuclear power plants. Implementations of the described principles were outlined. The described test machines are regularly used in the Paks Nuclear Power Plant, while the status and risk monitoring systems are now tested with field data and will soon be introduced in the plant. The concepts are constantly developed to make the test systems more generally applicable and effective.

References

1. C. V. Ramarmoorthy, et al. (1979) *A Systematic Approach to the Development and Validation of Critical Software for Nuclear Power Plants.* Proceedings of the 4th International Conference on Software Engineering, IEEE Computer Society Press, pp. 231–240.
2. B. Littlewood and D. Wright (1997) *Some conservative stopping rules for the operational testing of safety-critical software.* IEEE Transaction on Software Engineering, no. 23, vol. 11, pp. 673–683.
3. O. Schörner (1998) *Periodic Test Concept.* Siemens Kraftwerk Union AG, KWU NLL4 UNG 001/98/E006a, Cc. 2.04.1/03-2.

4. Varga, I. and Katics, B. (2002) *Testing concept for proper functioning of the reactor protecting system in the nuclear power plant of Paks.* Annals of DAAAM for 2002 and Proceedings of the 13th International DAAAM Symposium, pp. 587–588.
5. International Atomic Energy Agency (2002) *Instrumentation and Control Systems Important to Safety in Nuclear Power Plants.* IAEA Safety Standards Series, No. NS-G-1.3.
6. International Atomic Energy Agency (2002) *Solution for cost effective assessment of software based instrumentation and control systems in nuclear power plants.* IAEA Technical Documents, IAEA-TECDOC-1328.
7. Bokor, J., Szabó, G., Gáspár P., and Hetthésy J. (1997) *Reliability Analysis of Protection Systems in NPPs Using Fault-Tree Analysis Method.* In Proceedings of the IAEA Symposium on Computerized Reactor Protection and Safety Related Systems in Nuclear Power Plants, Budapest, 1997, pp 91–104.
8. Idaho National Engineering and Environmental Laboratory (1999) *Reliability Study: Combustion Engineering Reactor Protection System, 1984-1998.* INEEL, NUREG/CR-5500, Vol. 10, INEEL/EXT-97-00740.
9. A. Bondavalli and R. Filippini (2004) *Modeling and Analysis of a Scheduled Maintenance System: A DSPN Approach.* The Computer Journal, vol. 47, no. 6, pp. 634–650, 2004.
10. Varga, I., Bartha, T., Soumelidis, A., Katics, B. (2003) *A concept for on-line testing of distributed safety-critical supervisory systems.* IMS 2003, 7th IFAC Workshop on Intelligent Manufacturing Systems, Budapest, 2003, pp. 175–180.
11. Z. Simic, V. Mikulicic, I. Vrbanic (2001) *Risk Monitor Based on Risk Spectrum PSA Professional.* In Proceedings of the European Safety and Reliability Conference - ESREL, Turin, 2001. Available at http://www.aidic.it/italiano/congressi/esrel2001/webpapersesrel2001/157.pdf
12. Varga, I., Bartha, T., Szabó, G., Kiss, B. (2004) *Status and Actual Risk Monitoring in a NPP Reactor Protection System.* In Proceedings of the International Conference on Probabilistic Safety Assessment and Management (PSAM 7, ESREL '04), June 14–18, 2004, Berlin, vol. IV, pp. 1667–1682.

COMPAS – Compressed Test Pattern Sequencer for Scan Based Circuits

Ondřej Novák, Jiří Zahrádka, and Zdeněk Plíva

Technical University Liberec, Hálkova 6, 461 17 Liberec I, Czech Republic
ondrej.novak@vslib.cz

Abstract. This paper presents a software tool for test pattern compaction combined with compression of the test patterns to further reduce test data volume and time requirement. Usually the test set compaction is performed independently on test compression. We have implemented a test compaction and compression scheme that reorders test patterns previously generated in an ATPG in such a way that they are well suited for decompression. The compressed test sequence is decompressed in a scan chain. No design changes are required to be done in the functional part of the circuit. The tool is called COMPAS and it finds a sequence of overlapping patterns; each pattern detects a maximum number of circuit faults. Each pattern differs from the contiguous one in the first bit only, the remaining pattern bits are shifted for one position towards the last bit. The pattern first bits are stored in an external tester memory. The volume of stored data is substantially lower than in other comparable test pattern compression methods. The algorithm can be used for test data reduction in System on Chip testing using the IEEE P 1500 Standard extended by the RESPIN diagnostic access. Using this architecture the compressed test data are transmitted through a narrow test access mechanism from a tester to the tested SoC and the high volume decompressed test patterns are shifted through the high speed scan chains between the System on Chip (SoC) cores.

1 Introduction

Due to increased IC design complexity, utilization of sequential test patterns is almost completely abandoned and the full scan methodologies have become to be a standard. Nowadays we can see that these methodologies are not sufficiently effective. Higher circuit densities, larger number of embedded cores and long scan chains lead to non acceptable test data volume and testing time. New techniques are needed to reduce test data volume and testing time, as well as to overcome tester (ATE) bandwidth limitation.

Built-in pseudorandom or weighted random testing can be a solution of the above mentioned problems but still there remain random resistant faults, which should be tested from an ATE with deterministic patterns. Mixed-mode testing uses built-in pseudorandom pattern generators, which are usually used for generating first several thousands of test patterns (typically 10 000 patterns) and after the pseudorandom testing phase deterministic patterns are applied in order to test the random resistant faults. The deterministic patterns can be compressed, the decompression is usually done in the automaton that was used for pseudorandom test sequence generation; the seeds are stored in an ATE. Linear feedback shift register (LFSR) reseeding or output modification methods [11, 14, 18, 26,30] assume that a large portion of the bits in the test patterns are unspecified. The on-chip LFSR is seeded with such seeds that the bit sequence generated by the LFSR matches the deterministic patterns at the specified positions. The number of bits stored in a tester memory is then relatively small but the number of clock cycles, which is needed for testing, may be high. Random part of mixed-mode test is time and energy consuming. The hardware overhead may be also high because of difficult controlling the two test sessions.

As the ATPG test pattern generation has been able to keep up with increasing number of transistor counts for scan-based circuits, testing with deterministic patterns without any random test session is still actual. It spares testing time and the on chip hardware overhead is low. However the test sizes have been pushing test costs up due to the necessity of using more powerful ATEs and if the test access mechanism (TAM) is narrow the test application time becomes critical, too. In order to minimize the data transfer through the TAM, *compacted* and *compressed* test sets are used [9, 10]. By the term *compact test set* we mean a test set, which is created in the automatic test pattern generator (ATPG) from test patterns by merging as many as possible patterns. An original test pattern usually detects one or more possible circuit faults and contains several don’t care bits. The original patterns are merged in such a way that resulting patterns detect multiple faults and do not contain don’t care bits while the test set fault coverage remains unchanged.

Test data compression is a non-intrusive method that can be used to compress the pre-computed test set to a much smaller test set, which is then stored in the ATE memory. An on-chip decoder is used to generate the original test set from the compressed one. Many contributions containing different decompression mechanisms were published; let us mention the most recent ones: [1, 3, 5, 6, 8, 12, 17, 21, 22, 24 25, 28, 29]. It is not straightforward to compare the compression methods because some authors demonstrate the efficiency on decompression of random resistant faults only and other authors compress and decompress the whole ATPG deterministic test sequence. The usefulness of compressing algorithms and decompressing automata is influenced not only by the compression ratios but also by the complexities of the decompressing automata and by the computational complexities of the algorithms for finding the compressed test sequences.

In this paper, we present a software tool that prepares a sequence of pattern seeds to be stored in an ATE memory and decompressed with the help of a scan chain. The main idea is to maximally overlap the non compacted ATPG patterns. An example of decoding the test patterns from the ATE sequence is given in Fig. 1. The mentioned approach was firstly described in [7]. The compression method uses an algorithm for finding contiguous and consecutive scan chain vectors for the actual scan chain vector. These vectors are checked, whether they match with one or more remaining test patterns, which were previously generated and compacted with a help of some ATPG and which were not employed in the scan chain sequence yet. In [27] the compacted test vectors were reordered by a heuristic algorithm to attain maximal overlapping.

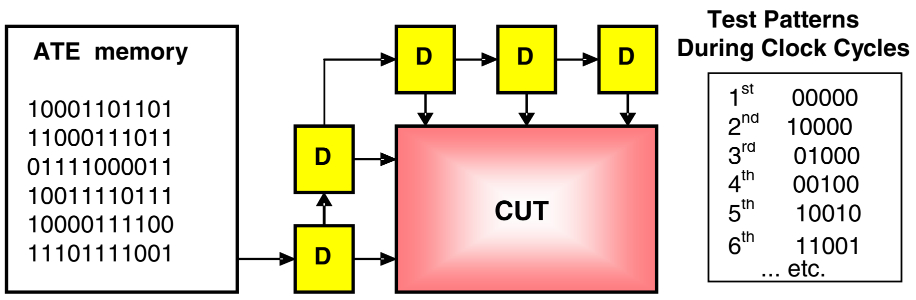


Fig. 1. Reusing of the scan chain flip-flops for test pattern decompression

We have performed several experiments with the above mentioned pattern compression principles. It was found that overlapping the compacted test patterns leads to longer test sequences then using non compacted test patterns with don't care bits. Further it was found that the algorithm for constructing the test sequence is extremely CPU time consuming for large circuits. Those observations have led us to propose an algorithm, which speeds up the computation by searching for the successors of the all zero seed only. The number of backtracks was reduced by application of this principle. The compression efficiency was improved by using non compacted test patterns and by fault simulation after every original ATPG test pattern application. In order to farther reduce the CPU time and to improve the efficiency, so called Future Array (FA) was introduced. In the FA those bits that are supposed to be generated later in the test sequence are remembered. Using this mechanism the number of computations needed for application of the patterns was reduced. The proposed algorithm spares a part of the CPU time, which is needed for test pattern generation, because the ATPG does not need to perform any fault simulation. The fault simulation is postponed and it is performed during test pattern overlapping. This improves the compression efficiency because of exploiting the interleaving patterns for testing the random testable faults.

The software tool that is based on the presented algorithm is called COMPAS (Compressed test Pattern Sequencer). It speeds up and improves the earlier algorithm [20] by taking into account possible future conflicts between overlapping patterns, it uses more efficient pattern coding and it remembers all information that could be useful in future algorithm loops. COMPAS is able to prepare test sequences for complex circuits in relatively short time. COMPAS can be also used for preparing test sequences for sequential cores under test (CUT) that are designed according the IEEE P1500 standard [18]. Test data can be effectively decompressed with the RESPIN test architecture [8]. This architecture reuses scan chains of different cores for updating the tested core scan chain content.

The rest of the paper is organized as follows. In Section 2 the test pattern compaction a compression algorithm is described. In Section 3 we demonstrate results of experiments with COMPAS and we compare the memory, time and clock cycles requirements with other methods. Finally, Section 4 concludes the paper.

2 Test Pattern Compaction and Compression in COMPAS

Let us suppose that we want to test a core under test (CUT), which is combinational and which is equipped with the boundary scan chain (SC) at first. The first part of the scan chain is formed by the input cells, the second part by the output cells. The primary outputs are connected to a space compactor and-or to a multiple input signature analyzer (MISR) so that it enables checking the test responses after every clock cycle. This arrangement enables us to test the CUT in the test-per-clock mode, the contiguous patterns are overlapped. Let us consider two contiguous patterns. Then the second pattern has the same content as the first pattern with the exception of the leftmost bit, which is obtained from the ATE. The remaining bits are shifted for one position to right. The algorithm calculates a sequence of the most left pattern bits that can be decompressed into patterns in the input part of the scan chain. The sequence should be as short as possible and the patterns should cover all faults of the CUT.

A Test Pattern List (TPL) together with the corresponding Undetected Fault List (UFL) should be prepared before running COMPAS. An ATPG tool that enables generating non-compacted test patterns should be used for this. At least one three state test vector with bit values 0, 1 and X (X means don't care value) should correspond with each considered fault. In this way it can be distinguished, which pattern should be deleted from the TPL after covering a fault from the UFL.

The main loop of the algorithm of finding the ATE memory bits is described in Fig.2. Let us suppose that the SC is reset before testing, which means that the all zero pattern is considered to be used as the first one. The fault coverage of this pattern is simulated and the detected faults are deleted from the UFL, test patterns corresponding to the detected faults are deleted from the TPL. Then the algorithm tries to compact the test set by overlapping remaining patterns with the actual vector. The algorithm searches, whether log 0 or log 1 is better to be used as the next actual (most left) chain bit. To do this, the algorithm finds positions of all patterns, in which the scan chain vector maximally overlaps the pattern and for which the actual bit to be introduced into the SC does not have a don't care value (see an example in Fig. 3). After finding the position the algorithm should count the usefulness U of the treated pattern. The pattern usefulness U is calculated according to the following formula:

$$U = c*(near\_to\_init + no\_dontcares)*inputs/2 + global\_dontcares$$

where:
near_to_init – the number of clock cycles that should be performed before placing the pattern into the scan chain
no_dontcares – the number of the don't care bits that are in the part of the pattern overlapping the scan chain
inputs – the number of CUT inputs
global_dontcare s– the total number of the pattern don't care bits
c –experimentally fixed parameter (implicitly $c = 1$)

The algorithm selects k patterns with minimum U (typically $k = 50$). Then the algorithm compares the number of the selected patterns with logical 1 on the actual position with the number of patterns with logical 0 on this position.

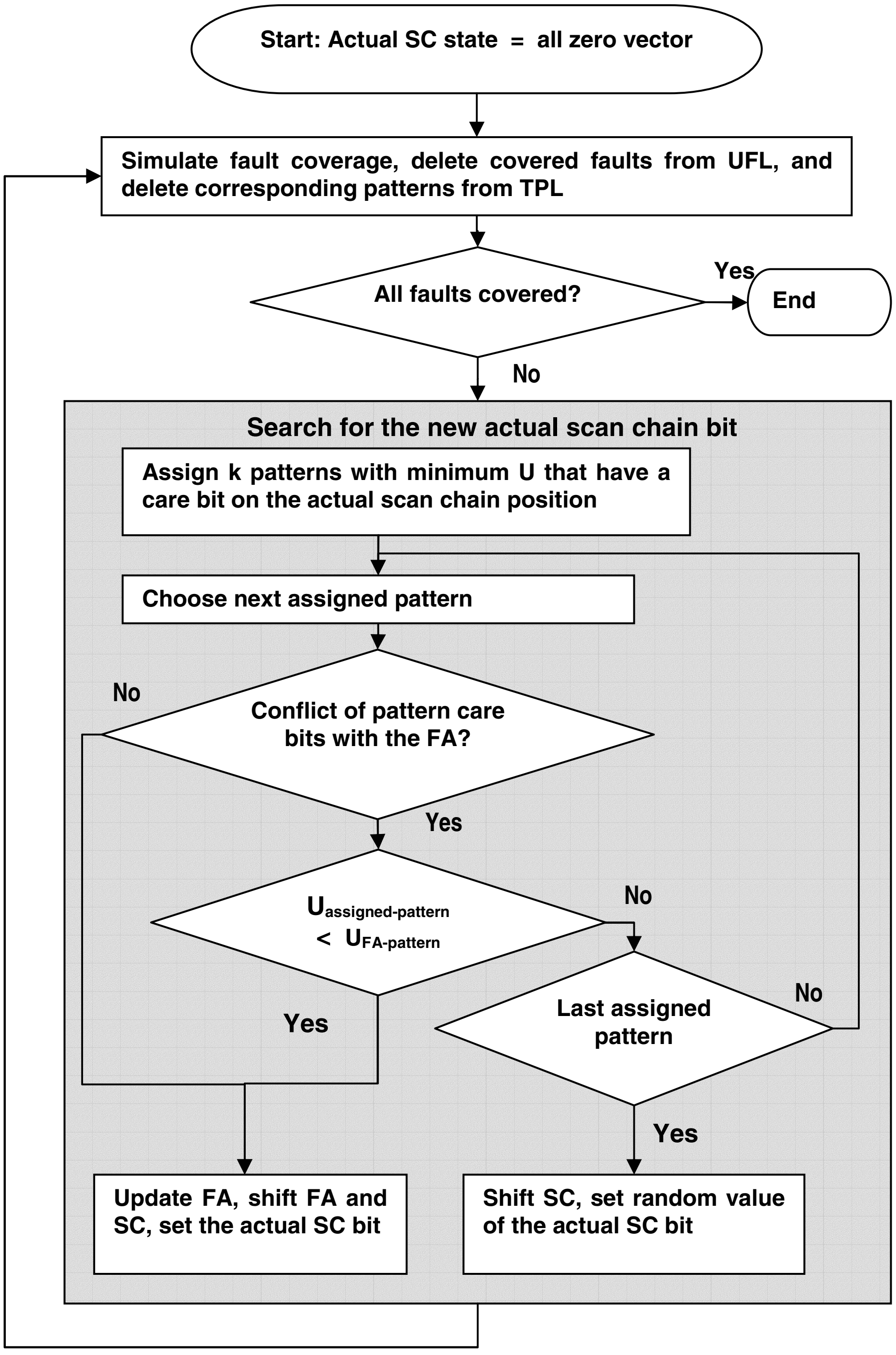

Start: Actual SC state = all zero vector
Simulate fault coverage, delete covered faults from UFL, and delete corresponding patterns from TPL
All faults covered?
Yes
End
No
Search for the new actual scan chain bit
Assign k patterns with minimum U that have a care bit on the actual scan chain position
Choose next assigned pattern
No
Conflict of pattern care bits with the FA?
Yes
U assigned-pattern < U FA-pattern
No
Yes
Last assigned pattern
No
Yes
Update FA, shift FA and SC, set the actual SC bit
Shift SC, set random value of the actual SC bit

Fig. 2. Algorithm of finding the input scan chain sequence

If the number of ones is greater than the number of zeros the patterns with logical one on the actual position are preferred in the future bit searching. (Application of this rule causes that the probability of more fault covering patterns construction is higher.) Starting with the first pattern the algorithm checks whether the patterns from the list of the best patterns matches with bits which will be necessary to be generated in the future clock cycles because of some previously selected patterns. These bits are stored in the Future Array (FA) together with their effectiveness and pattern identification information. If some position of FA is reserved for a logical value that is in conflict with the exercised pattern bit value, the algorithm compares the usefulness of both patterns and the winner is used in the future content of the FA, the other bit is deleted from the FA but the corresponding pattern is kept in the TPL. If no new test pattern can be applied a random value of the actual bit is set.

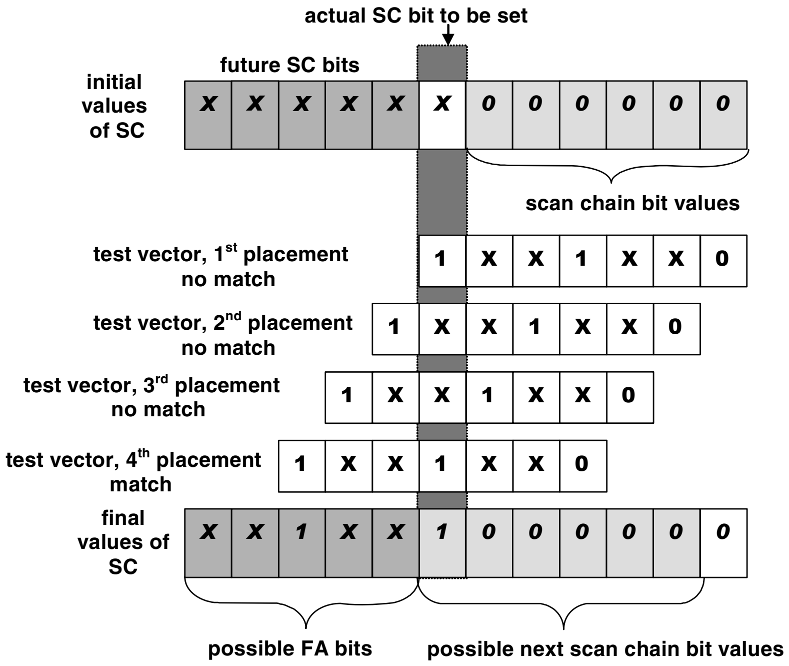


Fig. 3. Example of a test pattern matching: The SC initial value = (0,0,0,0,0,0). The test pattern (1,x,x,1,x,x,0) can be overlapped with the SC on the 4th placement. The possible final FA = (x,x,1,x,x) and SC = (1,0,0,0,0,0)

The fault simulation is performed after each new actual bit setting that causes that at least one original test pattern from the TPL is decompressed. The faults and patterns, which correspond to the covered faults, are removed from the lists. If there are no more faults in the UFL the algorithm terminates.

COMPAS system translates input test vectors into run length code format which minimizes the CPU time because of simpler manipulation with don't care bits. Great acceleration was obtained due to saving all intermediate results that could be used in the future computations. In the current version of COMPAS we use test patterns generated by the Atalanta ATPG tool [15] and the Hope fault simulation tool [16]. It runs either on the Unix or Windows platform. COMPAS is written in C++, consists of several routines, which can be run in parallel. The source code has approx. 2000 lines.

COMPAS can be used also for testing the scan based sequential circuits. It is necessary to avoid pattern overwriting by the previous test step responses. This can be done either by an additional hardware or by testing the scan chain in the test-per-scan mode, in which one test pattern is applied after n shifts, where n is equal to the scan chain length. In the test-per-scan mode we can use an auxiliary feedback shift register in which test patterns are kept during the previous test step response capturing. The next pattern can be reconstructed from the register content and one new bit from the memory with pattern seeds. This decompression mechanism can be done by the RESPIN architecture [8] that uses the scan chains of cores that are not under test for keeping the test patterns.

The COMPAS tool is on-line available on the http://iko.kes.vslib.cz/ web page. It compresses the test sets for the benchmark circuits and for user defined circuits described in the ISCAS89 format. The circuit should be described without the scan chain flip-flops, the scan chain flip-flops should be replaced by the primary outputs and inputs.

Table 1. Comparison of different test pattern compression techniques

Circuit name:		s13207	s15850	s38417	s38584
MinTest [3]	# of bits	163,100	58,656	113,152	161,040
Stat. Coding of Test Cubes [1]	# of bits	52,741	49,163	172,216	128,046
LFSR Reseeding [13]	# of bits	11,285	12,438	34,767	29,397
Illinois Scan Arch.[21]	# of bits	109,772	32,758	96,269	96,056
FDR Codes [6]	# of bits	30,880	26,000	93,466	77,812
Linear Decompressors [2]	# of bits	19,608	12,024	54,207	28,120
RESPIN ++ [26]	# of bits	26,004	32,226	89,132	63,232
COMPAS (proposed)	# of bits	4,024	7,737	21,798	6,675

3 Experimental Results

We have performed experiments with compaction and compression of the ISCAS89 circuits [1] test sequence by the COMPAS system. We have used a Pentium IV processor PC, 2.8 GHz, running under MS Windows XP. We have found that COMPAS is substantially less CPU time consuming than the previous versions of the compressing programs.

The most CPU time consuming circuit was the S38417 circuit. The compressed test sequence for this circuit was generated within 34 minutes. We have performed also several experiments with the ITC 99 benchmark circuits [12]. The largest circuit that was accepted by the ATPG Atalanta was the circuit b_17_opt_C; the CPU time spent by COMPAS was 5 hours for this circuit.

Table 1 shows the resulting numbers of stored bits for some well known test pattern compression methods and for the proposed algorithm. In the second column we plotted the test data volume for ATPG vectors, which were compacted only [3]. The third column shows the number of stored bits for statistical coding of the test patterns from the previous column [1]. Next results correspond to a combination of statistical coding and LFSR reseeding [13], compression with parallel/serial scan chains [21], frequency directed codes [6] and combinational linear decompression [2]. The column RESPIN++ shows the number of necessary bits given in [26]. These results correspond to mixed/mode testing scheme, 400 random test vectors generated within the RESPIN++ feedback registers were applied before decompression of deterministic patterns detecting the remaining faults. We have not included those methods of mixed-mode testing methods that use larger number of random patterns before testing by deterministic patterns as the resulting numbers of stored bits are not simply comparable.

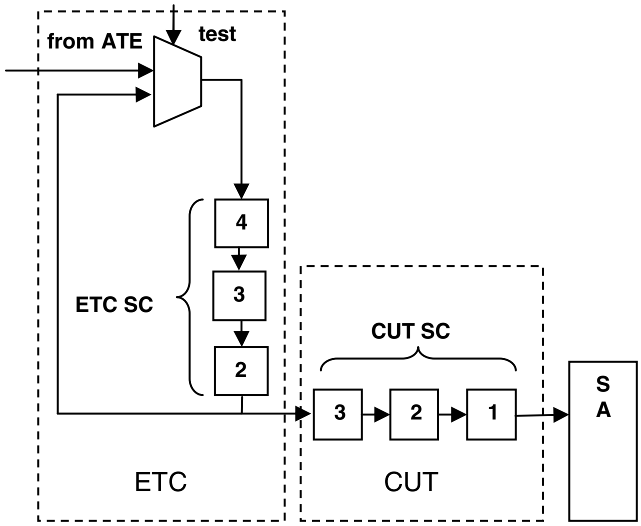


Fig. 4. An example of the RESPIN architecture with one scan chain in each core only. The 3 bit Embedded Tester Core (ETC) SC is connected by the wide TAM with the 3 bit Core under Test (CUT) SC. A narrow TAM connects a tester (ATE) memory, in which the compressed test sequence is stored, with the ETC. For decoding one pattern the following operations should be done: The multiplexer is switched so that the input from the ATE is active. The first bit of the ETC chain is loaded with the ATE bit; the rest of the ETC scan chain is shifted for one position. The ATE multiplexer is switched so that the feedback tap is active. Then two shifts are performed in the SCs. By performing the shifts a new test pattern is loaded into the CUT SC simultaneously with shifting the previous test step responses to the Signature Analyser (SA). After loading a pattern into the CUT SA the functional clock cycle is performed and the CUT flip-flops capture the test step responses

The table indicates that the number of bits, which are stored in a memory, could be substantially lower for the proposed method than for other pattern compressing methods. We have to admit that a fair comparison is conditioned by application of the same original test set. This was not the case of the presented table as we were not able to implement all other compression algorithms. Nevertheless considering that the ATPG tools could provide similar results, we can see that COMPAS provides extremely short test sequences. In order to enable more detailed comparison, we propose to all possible authors of compression algorithms to use the tools from the COMPAS web site.

We should note that the majority of the pattern compression methods do not use a fault simulation after encoding each new test pattern. In these methods the fault coverage was simulated during test pattern generation in the ATPG in the process of pattern compaction. The number of fault simulations in these cases corresponds to the total number of non compacted test patterns.

In case of COMPAS the ATPG patterns were generated without any simulation, fault simulation is performed after the pattern encoding. The number of fault simulations (No. of execute HOPE) and the CPU time spent on that (elapsed HOPE) can be checked for every circuit on the COMPAS Web page.

Another question is whether the test time of the proposed method is acceptable. For this reason we have compared the test time, which is necessary for testing the circuit designed with the RESPIN architecture with the COMPAS test sequence and the mixed-mode testing method [11] (Folding Counters).

In Fig. 4 we demonstrate an example of a simplified RESPIN architecture and the way of decompressing test patterns. The number of clock cycles that are needed for decompressing a test pattern is equal to the number of the CUT SC flip-flops, total number of clock cycles consists of clock cycles for decompression and a functional clock cycle. Usually the number of SCs is greater then one and the number of clock cycles for a pattern decompression is equal to the length of the longest SC in the CUT.

The method of Folding Counters uses a random pattern generator that generates 10,000 random test patterns that are pattern by pattern serially shifted into the scan chain and then the random resistant faults are tested by an output sequence of the Folding Counter. The counter is reseeded several times, in order to pass through the states that correspond to the deterministic test vectors. In order to keep a comparability of the Folding Counters with COMPAS, we consider that no pseudo input reduction is applied in both methods (no additional CUT dependent hardware is used inside the scan chain) and only one scan chain is used in one core.

Fig. 5 shows the results of the comparison for several ISCAS circuits. In case of Folding Counters we have added the number of clock cycles, which are necessary for random testing, with the clock cycles, which are necessary for rotating the folding counter after each reseeding, we obtain numbers of test clock cycles that are given in the graph.

The corresponding COMPAS curves show the number of clock cycles that should be performed in order to decompress the patterns if we consider one scan chain in one core only. We can see that COMPAS has lower numbers of clock cycles than the mixed mode testing approach [11]. Method [11] provides better results if additional CUT dependent SC branching and reconfiguration is accepted, COMPAS provides shorter tests for multiple scan chains. Other mixed-mode testing approaches have similar requirements on the number of clock cycles, so we may conclude that the proposed method is less time consuming than mixed-mode testing approaches.

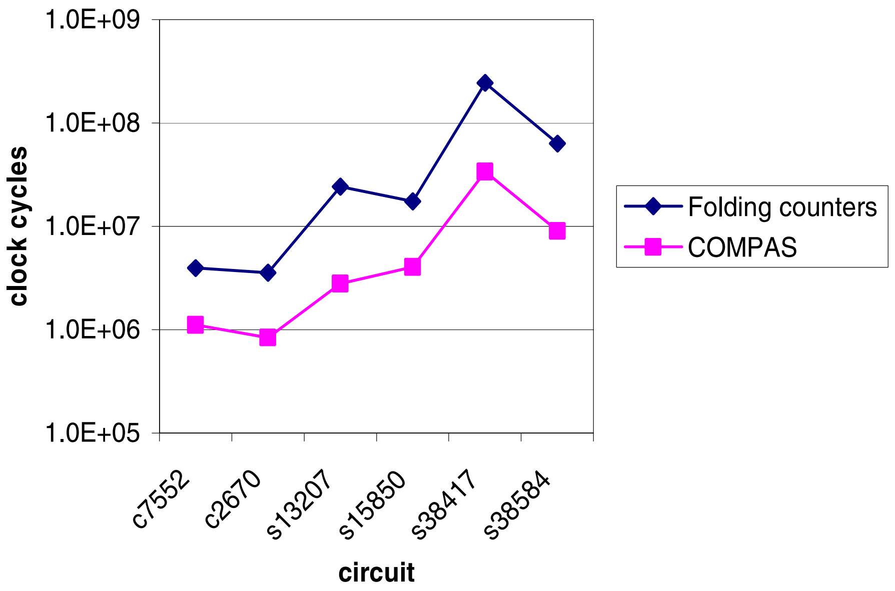


Fig. 5. Time for test application (given in number of clock cycles, which are necessary for test pattern loading and application)

4 Conclusion

The COMPAS compression tool demonstrates that it is possible to apply the method of test pattern compression through pattern overlapping for relatively large circuits and that the resulting test data volume is kept very low. COMPAS uses as input test patterns non compacted original ATPG test vectors with don't care bits. The patterns are overlapped and the resulting test sequence can be decompressed by the scan chain only. The decompressed patterns are simulated by the fault simulator whether they cover any other additional fault. This mechanism reduces the number of test patterns that should be used for testing since the interleaving patterns that appear in the scan chain between the original patterns cover the random testable faults. These faults are usually tested by the random pattern sequence in mixed-mode testing algorithms and the proposed method avoids using this random testing phase.

The proposed method of compression and compaction of test patterns is very well suited for testing combinational circuits with Boundary Scan because it does not require any additional diagnostic hardware for test pattern decompression. It can be used also for testing sequential cores with multiple scan chains. To do this we can use the RESPIN architecture. Following the IEEE P1500 standard [18] we do not require extra hardware with the exception of one multiplexer and a feedback wire in every core. The sequence generated by COMPAS can be used for less time consuming sequential core testing than it is possible by using known mixed-mode testing approaches.

Acknowledgement

The research was supported by the research grant of the Czech Grant Agency GACR 102/04/2137 and the grant IQS108040510 of the Czech Academy of Sciences.

References

1. Abhijit, J., Jayabrata G.D., and Touba, N.A.: Scan Vector Compression/Decompression Using Statistical Coding. Proc. of the IEEE VLSI Test Symposium ,1999
2. Bayraktaroglu, I., and Orailoglu, A.: Decompression Hardware Determination for Test Volume and Time Reduction through Unified Test Pattern Compaction and Compression. Proc. of the IEEE VLSI Test Symposium 2003
3. Barnhart, C., Brunkhorst, V., Distler, F., Farnsworth, O., Keller, B., Koenemann, B.: OPMISR: the foundation for compressed ATPG vectors. Proc. of the International Test Conference, 2001, pp. 748-157
4. Brglez, F., Bryan, D., and Kozminski, K.: Combinational Profiles of Sequential Benchmark Circuits. Proc. of. Int. Symp. on Circuits and Systems, 1989, pp. 1929-1934
5. Chandra, A., and Chakrabarty, K.: Test Data Compression for System-on-a-Chip Using Golomb Codes. Proc. of the IEEE VLSI Test Symposium, 2000
6. C Chandra, A., and Chakrabarty, K.: Frequency/Directed Run Length (FDR) Codes with Application to System/on/Chip Test Data Compression. Proc. of the IEEE VLSI Test Symposium 2001, pp. 42-47
7. Daen, W., and Mucha, J.: Hardware Test Pattern Generation for Built-in Testing. Proc. of the IEEE Test Conference, 1981, pp. 110-113
8. Dorsch, R. and Wunderlich, H-J.:Reusing Scan Chains for Test Pattern Decompression. Proc. of the European Test Workshop, 2001, pp. 24-32
9. Drineas, P., and Makris, Y.: Independent Test Sequence Compaction through Integer Programming. Proc of the ACM/IEEE International Conference on CAD, 2003
10. Hamazaoglu, I., and Patel, J.H.: Test Set Compaction Algorithms for Combinational Circuits. Proc. of the International Conference on Computer-Aided Design, 1998
11. Hellebrand, S., Liang, H.G., and Wunderlich, H.J.: A mixed mode BIST scheme based on reseeding of folding counters. Proc. of the International Test Conference, 2000
12. http://www.cerc.utexas.edu/itc99-benchmarks/bench.html
13. Krishna, C.V., Touba, N.A.: Reducing Test Data Volume Using LFSR Reseeding with Seed Compression. Proc. of the International Test Conference, 2002, pp. 321-330
14. Koenemann, B.: LFSR – coded test patterns for scan designs. Proc. of the European Test Conference, Munich , Germany, 1991, pp. 237-242
15. Lee, H. K. and Ha D. S.: On the generation of test patterns for combinational circuit. Technical Report 12_93, Department of Electrical Eng., Virginia Polytechnic Institute and State University
16. Lee, H. K. and Ha D. S.: HOPE: An efficient parallel fault simulator. Proc of the IEEE Design Automation Conference, pp. 336-340, June 1992.
17. Li, L and Chakrabarty, K.: Test Data Compression Using Dictionaries with Fixed-Length Indices. Proc. of the IEEE VLSI Test Symposium, 2003
18. Li, L. and Chakrabarty, K.: Test Set Embedding for Deterministic BIST Using a Reconfigurable Interconnection Network. IEEE Trans. on Computer Aided Design of ICs, Vol. 23, No. 9, Sept 2004, pp.1289-1305

19. Marinissen, E. J. Y., Zorian, R., Kapur, Taylor, T., and Whetsel L.: Towards a Standard for Embedded Core Test: An Example. Proc. of the IEEE International Test Conference , 1999, pp. 616–627
20. Novák, O., Nosek, J.: Test Pattern Decompression Using a Scan Chain. Proc. of the 2001 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems
21. Pandey, A. R. – Patel, H. J.: Reconfiguration Technique for Reducing Test Time and Test Data Volume in Illinois Scan Architecture Based Designs. Proc. of the IEEE VLSI Test Symposium, 2002, pp. 9-15
22. Rao, W., Bayraktaroglu, I. and Orailoglu, A.: Test Application Time and Volume Compression, Proc of DAC 2003, pp. 732-737
23. Rao, W., Oraiologlu, A.: Virtual Compression through Test Vector Stitching for Scan Based Designs. Proc. of the DATE conference, 2003
24. Rajski, J. et al.: Embedded deterministic test for low-cost manufacturing test. Proc. of the International Test Conference, 2002, pp. 301-310
25. Reddy, S.M., Miase, K., Kajihara, S., and Pomeranz, I.: On test data volume reduction for multiple scan chain designs. Proc. of the IEEE VLSI Test Symposium, 2002, pp. 103-108
26. Schafer, L., Dorsch, R., and Wunderlich, H.J.: RESPIN++- Deterministic Embedded Test. Proc. of the European Test Workshop, 2002, pp. 37-42
27. Su, C., Hwang, K.: A Serial Scan Test Vector Compression Methodology. Proc. of the International Test Conference 1993, pp. 981-988
28. Volkerink, E.H., Koche, A., and Mitra S.: Packet-based Input Test Data Compression Techniques. Proc. of the International Test Conference, 2002, pp. 154-163
29. Wolf, F. G., and Papachristou C.: Multiscan-based Test Compression and Hardware Decompression Using LZ77. Proc. of the International Test Conference, 2002, pp. 331-339
30. Wunderlich, H. J., and Kiefer G.: Bit-Flipping BIST, Proceedings ACM/IEEE International Conference on CAD-96 (ICCAD96), San Jose, California, November 1996, pp. 337-343

Fault Links: Exploring the Relationship Between Module and Fault Types

Jane Huffman Hayes[1], Inies Raphael C.M.[1], Vinod Kumar Surisetty[1], and Anneliese Andrews[2]

[1] University of Kentucky,
Computer Science Department
hayes@cs.uky.edu
[2] Washington State University,
School of Electrical Engineering and Computer Science
aandrews@eecs.wsu.edu

Abstract. Fault links represent relationships between the types of mistakes made and the type of module being developed or modified. The existence of such fault links can be used to guide code reviews, walkthroughs, allocation of verification and validation resources, testing of new code development, as well as code maintenance. We present an approach for categorizing code faults and code modules, and a means for examining relationships between the two. We successfully applied our approach to two case studies.

1 Introduction

As we seek to develop ever more complex systems, some with grave consequences of failure, we must strive to improve our technologies for developing and ensuring robust, reliable software. Fault-based analysis and fault-based testing are related technologies that seek to address this challenge.

Fault-based testing generates test data to demonstrate the absence of a set of pre-specified faults. Similarly, fault-based analysis identifies static techniques (such as traceability analysis that should be performed to ensure that a set of pre-specified faults do not exist. As part of fault-based analysis, a project manager can use historical data to determine what fault types are most likely to be introduced or can perform a risk analysis to determine what fault types would be most devastating if overlooked. Note that fault-based analysis is an early lifecycle approach that can be applied prior to implementation [15]. For example, developers of version 10 of a software system could use information on the number and type of faults from versions 8 and 9 to guide their code walkthroughs.

Based on our work on a semantic model of faults [30], Offutt's work on testing coupling [29], our work on traceability [16], and on requirement faults [15], we developed a conjecture about faults: The types of mistakes made by programmers largely depend on the type of module that is being developed or modified. We refer to this as a "fault link". A fault link is a relationship between the type of module being developed or changed and the fault type. For example, we posit that if a developer is writing a Computational-centric module, it is more likely that a computational fault will be introduced. Though this may seem intuitive or "not surprising," note that currently there are no empirical results to confirm it.

If we can demonstrate that fault links exist and if we can codify them, we can improve the development, testing, and maintenance of complex computer systems in several ways. We can offer preventative items for walkthrough checklists for newly developed code. We can recommend that exit criteria be added to walkthrough checklists for maintained code. For example, if a computational-centric module is being examined, do not exit the walkthrough until an extra check has been made to ensure that no computational errors exist. We can offer a list of fault-based tests that should be conducted based on the fault links. We can guide the allocation of verification and validation resources to best reduce risk.

The remainder of the paper is organized as follows. Section 2 presents related work. Sections 3 and 4 will present the module taxonomy and fault taxonomy, respectively. Section 5 discusses the research conjectures. Section 6 discusses two open source software case studies. We found evidence in favor of four of the conjectured fault links (as well as weak evidence for an additional two), such as Data-centric modules having many Data faults. We also found evidence of six unexpected fault links. Conclusions and future work are presented in Section 7.

2 Related Work

Faults have traditionally been characterized by syntactic categories [4, 22, 19], including where in the program the faults appear [17], which software development phase generated the faults [25, 20], what testing phase found the faults [30], and what type of statement or language feature the faults occur on [12]. As part of a NASA-funded project, Hayes has developed a taxonomy of requirements faults that is based on syntactic problems in the requirements [15].

A few attempts have been made to classify faults based on the mental mistakes that programmers make. IBM's ODC is one such scheme [18]. It assigns mental mistakes as part of a larger classification scheme.

Researchers have also examined change patterns of modules. Gall et al [13] used information about changes covering a sizeable number of releases to uncover logical dependencies and change patterns among modules. This was used to identify logical coupling among modules to uncover structural shortcomings. The work does not discriminate between corrective maintenance or enhancement related changes, thus did not attempt to classify faults. Similarly, Bieman et al [5] identified change-proneness of C++ code based on intentional use of patterns (or lack thereof). While this analysis found that some patterns are more change-prone in different categories of maintenance (corrective versus enhancement related changes), these faults were not classified. Bieman et al [6] found a strong relationship between class size and number of changes; larger classes changed more frequently. Also, classes that participate in design patterns and/or are reused through inheritance are more change-prone. They did not identify the type of change or fault in these studies.

Ohlsson et al [31] modeled fault proneness statistically over a series of releases. This included a variety of change measures at various levels of analysis, such as the number of defect fix reports attributed to a module, an interaction measure of defect repairs that involved more than one module, and impact of change measures (how many files affected, how many changes for each, various size of change measures by type of file). The analysis of the case study data showed that fault-prone modules showed higher system impact across four releases, where system impact is defined as total number of changes to .c and .h files in a release per module. This motivated construction of a fault architecture [24], which determines fault coupling and cohesion measures at the module and subsystem levels, within a release and across releases. Nikora and Munson presented a predictor for fault prone modules. They used a set of metrics and a reduced set of domains to build their predictor. They did not classify faults though and did not classify modules beyond "fault prone" or not "fault prone [28]."

Ostrand et al [32], with the aim of aiding organizations to determine the optimal use of their testing resources, have identified various file characteristics. These characteristics can serve as predictors of fault-proneness. By examining a series of 13 releases of a large evolving industrial software system, they observed that: (i) faults are concentrated in small numbers of files and in small percentages of code mass, (ii) shortchanging the testing efforts for previously high-fault files is a mistake, and (iii) "all late-pre-release faults always appeared in under 5% of the files"[32].

However, no effort was made to classify modules and faults. Fenton et al [11] have quantitatively analyzed the faults and failures of a major commercial system. Some of their observations were identical to those made by Ostrand et al [32]. Fenton et al provided strong evidence to suggest that software systems that are developed under the same environment result in similar fault densities, when tested in similar testing phases.

3 Module Taxonomy

Any simple or complex program can be viewed as a combination of various modules. A module is just a part of a program, which aids in performing some action or in making decisions to perform actions. A module can be a single statement or a single function or procedure that contributes to the purpose of the program.
We identified two methods for categorizing modules by type:

- Method one: Program modules are classified based on their main purpose. We considered allowing modules to have a second category based on their secondary purpose, but decided against it for the present. This represents a possible area for future work. This method is easy to comprehend and apply and is also faster than method two. However, it does not easily lend itself to automation.
- Method two: Modules are classified based on the percentage of lines of code that perform specific functions, such as computation, data manipulations, etc. We count the number of lines that belong to a particular category in a module, select the category with the highest Lines Of Code, and assign the module to that category. For example, "IF (salary > 1000)" is a controller statement. This method provides

information about the statements used in a program and is easily automated with some standard guidelines. Unfortunately, there are drawbacks including: (i) difficult to perform categorization, (ii) time consuming, (iii) tedious when performed manually, and (iv) not easy to understand.

This paper classifies modules using method one. We followed a subset of the steps in [15] to develop module and code fault taxonomies: select a fault taxonomy as the basis for the work, examine sample code faults, adopt or build a method for extending the fault taxonomy, and implement the method for tailoring a taxonomy.

Our original module and fault taxonomy was influenced by the prior work discussed in Sections 2 and 4. We also performed a pilot study on an industrial partner's project as well as on student programming assignments to further construct the taxonomies. We applied the two taxonomies to categorize the faults and modules in two open source web-based projects, and detected new categories for both the fault and module taxonomies. Two new module categories were added including error handling and environmental setup. Fig. 1 shows the resulting generic module taxonomy. It is applicable to most programs and domains, but could be tailored to a specific domain or application using the process in [15]. Each module category is described below.

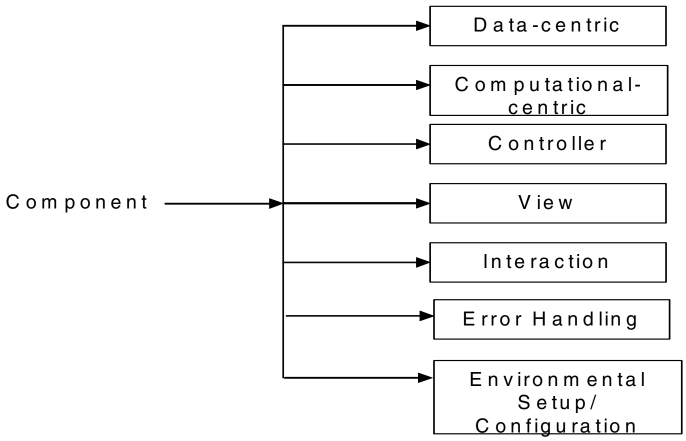


Fig. 1. Taxonomy of program modules

- Data-centric: Modules that deal with data definition and handling fall under this category. Access to database is also classified under data centric module.
- Error Handling: The main purpose of modules in this category is to handle exceptions or errors that are likely to occur.
- Computational-centric: Modules whose main purpose is to calculate or compute results belong in this category. At the statement level, any statement that changes any variable or state of the program falls under this category.
- Controller: Any module whose main purpose is to control the sequence of program execution falls under this category.

- Environmental setup/configuration: The main purpose of the modules is to set up an appropriate environment for the software to function efficiently.
- View: Any module that designs or handles graphical user-interface controls or manipulates the attributes of the controls is part of this category. Also, the statements used for displaying information belong to this category.
- Interaction: Any module or statement that performs a function call or passes parameters to other modules or tries to access the data structures outside the module falls under this category.

4 Fault Taxonomy

Our fault taxonomy does not include errors that can be caught by the compiler at compile time. We attempted to make the module and fault taxonomies generic enough to be language independent and method independent. Fig. 3 presents a graphical depiction of the taxonomy. The branches of the tree represent fault categories that are language independent, but the leaves may be language dependent. For example, the control/logic fault type applies to any language but register reuse will only be applicable for languages such as C or assembly languages.

The fault taxonomy also takes practical realities into account. Specifically, the taxonomy only relies on bug reports or problem reports and does not assume that (up to date) specifications or design are available for analysis. The following fault types are significant and have been included because they have been shown to be important fault categories in the past [3, 9, 10, 14, 21, 23, 35, 36].

Data: *Incorrect data definition.* Data definition involves assigning a name, type, and size for a data item. Since some data types are compatible with others (e.g., float can take an integer value), misuse can result in errors that are not detected at compile time. *Improper data initialization* is caused by the failure to initialize or reinitialize a data structure properly upon module entry or exit [3]. Examples of this include control blocks, registers, or switches not cleared or reset before transition [10]. *Improper data representation.* By representation we mean the ways in which the data is stored, i.e., data structure. The information or data can be stored in different ways, e.g., structured as a database or unstructured in flat files. Program statements that don't properly account for data representation may compile, but could result in runtime problems.

Computational: Errors that lead to a wrong value being calculated for a variable or register or switch belong in this class.

Control/logic: "Errors that cause an incorrect path in a module to be taken are considered control errors [3]." We group logic errors here also. *Statement logic* [36] faults cause the executable statements to be executed in the wrong order or not at all. For example, a program may fail to perform validation before returning the data. *Sequence errors* [36] exist when the order in which messages and control information are sent is erroneous. For example, the server program in a client-server environment may send an acknowledgment without receiving any request from the client. *Unreachable code* [25] occurs due to errors in control or logic statements. *Performance* faults may affect the overall performance of the software.

Interface: Here we include "errors associated with structures existing outside the module's local environment but which the module used" [3] and errors in the communications between modules. For example, incorrect subroutine or module call, insufficient data transfer [25], "incorrect declaration of COMMON segment" [3] all fall under this category.

User interface: Faults that interfere with the efficiency, performance and appearance of the user interface of the software. *Large response time* [23, 35] causes the interface controls to respond with delay. *Lack of naturalness* [21] is caused by a number of factors such as illogical grouping of information, use of uppercase, use of arbitrary abbreviations, etc. A natural interface does not cause the user to significantly alter his or her approach to the task in order to interact with the system. *Inconsistency* [35, 21, 14] refers to the lack of a pattern of familiarity designed throughout a product. *Redundancy* [21] in a user interface requires the user to enter unnecessary information for an operation. For example, a user should never have to supply leading zeros ("00090.45" instead of "90.45"). *Complexity* [35] leads to interfaces that are not simple and easy to work with. The interface must be simple. The complexity of a user interface is based on the following factors**:** ease of use, ease of learning and understanding, and ease of navigation. *Lack of support* [21] refers to the limited amount of assistance the interface provides to the user. *Not flexible* [21, 14] refers to a user interface that narrows the types of users that can work on the software. The user interface must be able to tolerate different levels of user familiarity and performance. *Unpredictable flow* is when the flow of control in the user-interface gets beyond the scope of the user. An example of unpredictable flow is when the user tries to perform a spell check on her document and the software also performs a thesaurus function, despite not being invoked by the user. *Visual stimulation* [35, 21] refers to faults dealing with the improper use of color, fonts, graphics, control layout, etc. The determination that a fault exists is based on a bug report. Thus, we do not need to define metrics to measure attributes like "ease of use" or "ease of navigation".

Framework [9]: There are certain languages that make use of the concept of packages or reusable code. In such a language, a particular program imports or includes some of the packages to avoid unnecessary work. The set of statements used for this purpose is classified as "framework." *Missing framework elements* are caused when, upon integration, some modules might not have included required setup files. When all the modules are compiled together or individually, the compiler does not show any errors. However, at run time when the module calls or tries to communicate with another module, an error occurs. As mentioned before, only the leaves of the classification tree may be language dependent. Thus the fact that a framework element is missing is language independent, while the specifics of element mismatch will be language dependent.

5 Research Conjectures

After developing the fault taxonomy and module taxonomy, we noticed a strong correspondence between the categories, resulting in the following research question: "Does the module type drive the fault type one encounters?" We developed several research conjectures about fault links based on this. They are justified by prior work and the pilot studies mentioned earlier. The following 10 fault links were posited:

C1.1 – Data-centric modules will have a higher percentage of Data faults.
C1.2 - Data faults will occur more frequently in Data-centric modules.
C2.1 – Controller modules will have a higher percentage of Control/Logic faults.
C2.2 - Control/Logic faults will occur more frequently in Controller modules.
C3.1 – Computational-centric modules show a high percentage of Computation faults.
C3.2 – Computation faults occur more frequently in Computational-centric modules.

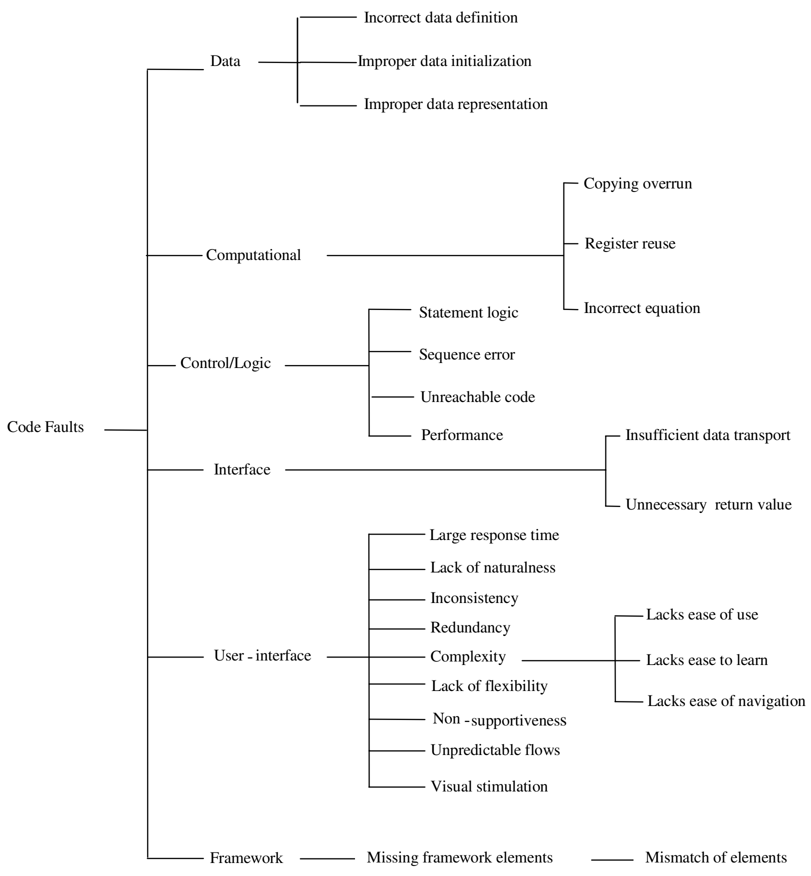


Fig. 3. Fault Taxonomy

C4.1 - Interaction modules will have a higher percentage of Interface faults.
C4.2 – Interface faults will occur more frequently in Interaction modules.
C5.1 – View modules will have a higher percentage of User Interface faults.
C5.2 – User Interface faults will occur more frequently in View modules.

We also posited secondary research conjectures. These are not as intuitive as the above, and some counter the above conjectures.

C6.1 – Interaction modules will have a higher percentage of User Interface faults.
C6.2 – User Interface faults will occur more frequently in Interaction modules.
C7.1 – View modules will have a higher percentage of Framework faults.
C7.2 – Framework faults will occur more frequently in View modules.
C8.1 – Error Handling modules will have a higher percentage of Data faults.
C8.2 – Data faults will occur more frequently in Error Handling modules.
C9.1 – Environmental Setup/Configuration modules will have a higher percentage of framework faults.
C9.2 – Framework fault occur more frequently in Environmental Setup/Configuration modules.

6 Case Studies

We applied the taxonomy to two open source software systems, Apache and Mozilla, to evaluate whether this taxonomy can be applied to common types of software and to see whether bug reports typical for such applications are able to reveal enough information. Both systems are largely written in C/C++. The Apache sever is a powerful, flexible, HTTP/1.1-compliant web server [2]. Mozilla is an open-source web browser, designed for standards compliance, performance, and portability [26]. Bugzilla is a "Bug-Tracking Systems" used in the Mozilla project. It allows individual or groups of developers to effectively keep track of outstanding bugs in their product.

6.1 Apache Case Study (Modules and Faults)

We examined all 30 modules that existed at the time of the study (100%). The size of the modules ranged from 250 LOC to 4500 LOC. We randomly selected two releases for which to examine bug reports, releases for the years 1999 and 2000. For those years, there were 2300 bug reports. Of these, we examined 300 (13%). Of these 300, only 177 bug reports provided enough information for fault categorization.

We classified all modules of the Apache 1.3.24 server [2] based on module purpose (method one). Table 1 presents the distribution of the Apache module classification. In this table, the percentage column denotes the percent number of modules of a particular type. For example, 10% of the 30 (3) modules were categorized as View modules. The largest module categories were Controller and Computational-centric, at 26.7% each.

Next, we applied method two to classify a subset of the Apache modules. Though many modules were categorized as belonging to the same categories when using method one and method two, some modules were not. For example, the following module was categorized as Computational-centric using method one, but was typed as Controller using method two.

Module:	mod_unique_id
Main purpose:	generate unique request identifier for every request
Method one:	classified as computational module

Method two: As you can see from Table 2, the number of lines of code performing control/logic (Controller) functions is greater than the number of lines performing other functions. Therefore it is categorized as Controller.

The advantage of method two is that categorization can be automated. However, the results of method two are not always intuitive. Method one is more subjective than method two. While subjective measurement can and should be systematic, it lacks the rigor of objectively measurable and quantitative scales [38]. To account for this, one normally develops reliability indicators for such scales (for example, inter-rater reliability) [1]. To that end, we performed an inter-rater reliability survey. We had five software engineers apply method one to this same module. The engineers were given the code for the module (including in-line documentation) and a list of and definitions for the module types in our taxonomy. All five engineers labeled the module as "computational." This convinced us that our subjective method exhibits reliability, so we continued using the results from method one.

Because of our interest in the relationship between module type and fault type, we performed a second step for the Apache case study. We went back to the 30 modules we had categorized and attempted to locate bug reports or problem reports for each. The problem reports provide information on identified faults. These have not necessarily been fixed. Several hundred bug reports were listed for each module. We examined a subset of these bug reports for a subset of the 30 modules (cf. Table 3).

Some general observations can be made. Many bug reports did not document bugs. Some bug reports represented enhancement requests. Bug reports had been generated by users who were "just trying out the bug tracking system." Many bug reports did not relate to code faults, but to poor documentation. Some bug reports did not relate to the version of Apache that we were examining or did not state the version number. Bug reports were duplicated or not deemed errors by the Apache developers. Finally, many bug reports documented more than one code fault and should have been separated into multiple bug reports. On average there were 1.5 faults per bug report.

We adjusted our approach to accommodate these findings. We first weeded out the "non-bug reports." Next, we disregarded bug reports not related to code. We then eliminated bug reports that did not relate to version 1.3.24 of Apache or were not actual errors per the Apache engineers. We then examined each fault in isolation, even if several had been grouped in one bug report. As we did not examine the same number of modules of each type (e.g., we examined eight Computational-centric, but only two Interaction modules), we looked at the faults as a function of the number of faults per module. That is, we examined 33 faults for four Data-centric modules. The 33 faults were categorized according to the fault taxonomy. The resulting values were scaled to reflect 8.25 faults per module.

Table 3 shows the module and fault classification for the Apache study. Module types are shown in the rows. The columns indicate: the total number of modules of different types that were examined; the number of faults, by fault type, for each module type; the total number of faults for the module type; and the percentage of faults found in a particular module type. For example, the Controller module row indicates that six such modules were examined, that 28 Control/Logic faults were found in the Controller modules, that a total of 47 faults were found in controller modules accounting for 26.6% of all faults classified. The highest value in each row has been bolded, and the highest value in each column has been italicized. In the above example, the value “28” has been bolded and italicized as it is the highest value for both the row and column. The bottom row indicates the percentage of each fault type classified. For example, 91 Control/Logic faults were found and they accounted for 51.5% of all faults.

It is clear that control/logic faults dominate this case study, regardless of module type. Though we had not conjectured this, it is not such a surprising result. In our own experience as programmers, teachers, and lab assistants for junior level programming courses, we have also noticed that these errors dominate.

Table 1. Classification of Apache Modules by Type

Module	Number	Percentage
Data-centric	6	20 %
Controller	8	**26.7%**
Computational-centric	8	**26.7%**
View	3	10%
Interaction	3	10%
Error Handling	1	3.3%
Environmental Setup/Configuration	1	3.3%
Total	30	100%

Table 2. Mod_Unique_ID Categorization-Method Two

Module	LOC (Lines Of Code)
Data-centric	62
Controller	**68**
Computational-centric	12
View	0
Interaction	11
Error Handling	0
Environmental Setup/Configuration	0
Total	153

Table 4 illustrates the “have” relationship that exists between the module and fault types. For example, a Data-centric module has more Control/Logic faults than any other type of fault, and these account for 48% of the faults typically found in a Data-centric module. The module types are listed in the rows of the table. For ease of illustration, two columns (the total number of modules of different types that were examined and the total number of faults grouped by the module in which they occur) have been repeated here from Table 3. We also show the total faults per module, followed by the percentage of faults found in a particular module type. Note that changes have been made to the values in the module-fault cells. Each cell has two values, a percentage value and a fault-per-module value. The percentage value represents the “have” relationship. The fault-per-module value indicates that out of N total faults in a module of a particular type, X of them belongs to a particular fault type. For example, let us examine the row for the Data-centric module. The number of data-centric modules examined was four, the total number of faults from the four modules was 33, and therefore the total faults per module (33/4) is 8.25 per module. This total fault-per-module value is distributed across the fault types based on their count from Table 1. As far as the fault distribution across different fault types is concerned, the data-centric module had about 18% data faults, 48% C/L faults, 6% computational faults, 9% interface faults, 18% framework faults, and zero percent GUI faults. As before, the highest value in each row and column is bolded and italicized respectively.

Table 5 illustrates the "occurs-in" relationship that exists between the fault and module types. For example, Data faults tend to occur in Data-centric modules most frequently (46%). The table is very similar to Table 4 except that it illustrates the "occurs-in" relationship from the fault type to the module type. The cells have the same two types of values as before, the percentage value and fault-per-module value. The fault-per-module value has the as meaning as before, but the percentage value in this case represents the "occurs-in" relationship. For example, let us examine the row for the Control/Logic fault. We can see that 16.8% of the C/L faults occur in data-centric modules, 19.6% of the C/L faults occur in controller modules, 11% of the C/L faults occur in computational-centric modules, 14.7% of the C/L faults occur in interaction modules, 21% of the C/L faults occur in view modules, 12.6% of the C/L faults occur in error-handling modules, and 4% of the C/L faults occur in environmental setup modules. The total faults-per-module and faults-per-module values of each are calculated as before.

Next, we assess the "have" relationship (from Table 4). The most frequently occurring fault type in Data-centric modules was Control/Logic at 48% (no close second). This does not support C1.1. The most frequently occurring fault type in Controller modules was Control/Logic at 59.6% with no close second. This does strongly support C2.1. The most frequently occurring fault type in Computational-centric modules was Control/Logic at 48% with no close second. This does not support C3.1. The most frequently occurring fault type in Interaction modules was Control/Logic at 41% with no close second. This does not support C4.1 or C6.1. The most frequently occurring fault type in View modules was Control/Logic at 50% with no close second. This does not support C5.1 or C7.1. The most frequently occurring fault type in Error Handling modules was Control/Logic at 75%. This does not support C8.1. There was a tie for most frequently occurring fault type in Environmental Setup/Configuration modules, 50% for both Control/Logic and Computational (no support for C9.1).

Table 5 illustrates the "occurs-in" relationship that exists between the fault and module types. For example, Data faults tend to occur in Data-centric modules most frequently (46%). The table is very similar to Table 4 except that it illustrates the "occurs-in" relationship from the fault type to the module type. The cells have the same two types of values as before, the percentage value and fault-per-module value. The fault-per-module value has the as meaning as before, but the percentage value in this case represents the "occurs-in" relationship. For example, let us examine the row for the Control/Logic fault. We can see that 16.8% of the C/L faults occur in data-centric modules, 19.6% of the C/L faults occur in controller modules, 11% of the C/L faults occur in computational-centric modules, 14.7% of the C/L faults occur in interaction modules, 21% of the C/L faults occur in view modules, 12.6% of the C/L faults occur

in error-handling modules, and 4% of the C/L faults occur in environmental setup modules. The total faults-per-module and faults-per-module values of each are calculated as before.

Next, we assess the “have” relationship (from Table 4). The most frequently occurring fault type in Data-centric modules was Control/Logic at 48% (no close second). This does not support C1.1. The most frequently occurring fault type in Con troller modules was Control/Logic at 59.6% with no close second. This does strongly support C2.1. The most frequently occurring fault type in Computational-centric modules was Control/Logic at 48% with no close second. This does not support C3.1. The most frequently occurring fault type in Interaction modules was Control/Logic at 41% with no close second. This does not support C4.1 or C6.1. The most frequently occurring fault type in View modules was Control/Logic at 50% with no close second. This does not support C5.1 or C7.1. The most frequently occurring fault type in Error Handling modules was Control/Logic at 75%. This does not support C8.1.

There was a tie for most frequently occurring fault type in Environmental Setup/Configuration modules, 50% for both Control/Logic and Computational. This does not support C9.1.

Table 3. Module and Fault Type Classification for Apache Study

Module type	# modules	Fault type						Total Faults	%
		Data	C/L	Comput.	Interface	Framework	GUI		
Data-centric	4	*6*	**16**	2	3	*6*	0	33	18.7%
Controller	6	3	***28***	5	4	5	*2*	47	26.6%
Computational-centric	8	2	**21**	*7*	*7*	*6*	1	44	24.8%
Interaction	2	0	**7**	3	3	2	*2*	17	9.6%
View	3	3	**15**	4	1	5	2	30	17%
Error Handling	1	0	**3**	1	0	0	0	4	2.2%
Environ. Setup	1	0	**1**	**1**	0	0	0	2	1.1%
Total	25	14	91	23	18	24	7	177	100%
Percentage		8%	51.5%	13%	10%	13.5%	4%	100%	

As can be seen from Table 5 (the “occurs-in” relationship), the majority of the Data faults occur in the Data-centric modules (46%). The next highest value is 30.7% for View modules. This finding provides support for C1.2, but not for C8.2. The majority of Control/Logic faults occur in View modules (21%) with Controller modules bringing up a close second at 19.6%. This finding lends some support to C2.2, but not as strong as for C1.2. Computation faults occur 36% of the time in Interaction modules followed by View modules at 19%. This does not support C3.2. Interface faults accounted for 36% of the Interaction module faults with no close second. This strongly supports C4.2. The majority of Framework faults occurred in View modules (29%) with Data-centric modules close behind at 26%. This provides some support for C7.2, but not C9.2. 47% of the User Interface faults occur in Interaction modules (supports C6.2, but not C5.2).

Our findings are summarized in Table 6. The basic question was: "Does the module type drive the fault type?" Six conjectured fault links were supported, at least weakly. Thus we found evidence for answering "yes." A fault link that appeared universally, though not conjectured, was Control/Logic faults being the most prominent fault type for all module types. One could view this as an additional six fault links (data modules have Control/Logic (C/L) faults, computational-centric modules have C/L faults, Interaction modules have C/L faults, View, Error Handling, and Environment Setup/ Configuration modules have C/L faults). This finding would lead one to answer the overarching question "no." Our results are still inconclusive, but appear to hold promise.

6.2 Mozilla Case Study (Faults and Modules)

Next, we examined problem reports for the open source software product Mozilla (web browser) using the bug tracking system Bugzilla [26]. Mozilla is a very large software system and provided a plethora of problem reports for sampling. We examined 70 bug reports, selected randomly using Bugzilla. From these, 75 faults were identified that were code-related. Note that the "fault per problem report" ratio was only 1.07 as compared to 1.5 for Apache. These faults were categorized using our fault taxonomy. Table 7 presents the high level distribution of the faults found in Mozilla. 53.4% of faults reported for the open source software Mozilla fall under the category of Control/Logic faults, reinforcing findings from the first case study.

We were not able to find the modules that tied to specific bug reports or vice versa, as we were able to do in Apache. So we next randomly selected 30 modules in the Mozilla directories and categorized them. As can be seen from Table 8, the majority of the modules fell under the category of Computational-centric (26.7%), with Controller just behind at 20%. This is consistent with our findings for the Apache study.

6.3 Comparison of Case Study

Both case studies exhibit strong similarities with regard to fault types and module types. For both systems, Control/Logic faults occurred most frequently: 50% for Apache and 53.4% for Mozilla. The next most frequent fault type for Apache was a tie between Interface and Framework at 14%. For Mozilla, it was Data at 17.3%. The fourth most frequent fault type for Apache was Data at 10%, and it was a three-way tie for Mozilla between Computational, Interface, and User Interface, all at 8%. A striking result was the dominance of the Control/Logic fault type, in both systems.

The most frequent module type for Mozilla was Computational-centric at 26.7%. Computational-centric was tied for most frequent with Controller at 26.7% for Apache. The next most frequent module type for Apache was Data-centric at 20%. For Mozilla, it was Controller at 20%. The third most frequent module type for Apache was a tie between View and Interaction, both at 10%. For Mozilla, Data-centric and View tied for 16.7%. Computational-centric and Controller occurred most frequently in both systems. A comparison of fault type percentages is shown in Fig. 2. In each category, the percent of faults in the two applications are similar (note that Apache does not report user interface bugs, since it is not interactive). For the common fault types, correlation analysis found a correlation value of 0.94 between the faults percentages. This is not surprising, as these applications share common characteristics: open source, web related. The result also confirms that our fault taxonomy is reasonable and applicable.

Table 4. "Have" Relationship from Module to Fault Types for the Apache Study

Module type	# modules	Fault type						Total Faults	Total Faults/ module	%
		Data	C/L	Comput.	Interface	Frame-work	GUI			
Data-centric	4	*18%* 1.5	**48%** 4	6% 0.5	9% 0.75	*18%* 1.5	0% 0	33	8.25	18%
Controller	6	6% 0.5	**59.6** % 4.67	10.6% 0.83	8.5% 0.67	10.6% 0.83	4% 0.33	47	7.83	17%
Computat.-centric	8	4.5% 0.25	**48%** 2.63	16% 0.875	16% 0.875	13.6% 0.75	2% 0.12	44	5.5	12%
Interaction	2	0% 0	**41%** 3.5	17.6% 1.5	*17.6%* 1.5	11.7% 1	*11.7%* 1	17	8.5	18%
View	3	10% 1	**50%** 5	13% 1.33	3% 0.33	16.7% 1.67	6.7% 0.67	30	10	21.8%
Error Han-dling	1	0% 0	***75%*** 3	25% 1	0% 0	0% 0	0% 0	4	4	8.7%
Environ. Setup	1	0% 0	**50%** 1	***50%*** 1	0% 0	0% 0	0% 0	2	2	4.5%
Total	25	3.25	23.8	7.035	4.125	5.75	2.12		[46.08, 46.08]	100%

Table 5. The “occurs-in” Relationship from Fault to Module Types for the Apache Study

Fault Type	Module Type							Total	%
	Data-centric	Controller	Computat.-centric	Interaction	View	Error Handling	Environmental setup		
#modules	4	**6**	8	2	3	1	1	25	
Data	***46%*** 1.5	15% 0.5	7.7% 0.25	0% 0	30.7% 1	0% 0	0% 0	3.25	10%
C/L	16.8% 4	*19.6%* 4.67	11% 2.63	14.7% 3.5	**21%** 5	12.6% 3	4% 1	23.8	50%
Computational	7% 0.5	11.7% 0.83	12% 0.875	**36%** 1.5	19% 1.33	*14%* 1	*14%* 1	7.035	9%
Interface	18% 0.75	16% 0.67	*21%* 0.875	**36%** 1.5	8% 0.33	0% 0	0% 0	4.125	14%
Framework	26% 1.5	14% 0.83	13% 0.75	17% 1	**29%** 1.67	0% 0	0% 0	5.75	14%
GUI	0% 0	15.5% 0.33	5.6% 0.12	***47%*** 1	*31.6%* 0.67	0% 0	0% 0	2.12	3%
Total faults	33	47	44	17	30	4	2	177	100%
Total Faults/module	8.25	7.83	5.5	8.5	10	4	2	[46.08, 46.08]	

Table 6. Conjecture Results

Conjecture	Conjectured Fault Link	Supported?
C1.1	Data modules have Data faults	No
C1.2	**Data faults occur in Data modules**	**Yes**
C2.1	**Controller modules have C/L faults**	**Yes**
C2.2	**C/L faults occur in Controller modules**	**Weak**
C3.1	Comp. modules have computational faults	No
C3.2	Comput. faults occur in Comput. modules	No
C4.1	Interaction modules have Interface faults	No
C4.2	**Interface faults occur in Interaction modules**	**Yes**
C5.1	View modules have User Interface faults	No
C5.2	User Interface faults occur in View modules	No
C6.1	Interaction modules have User Interface faults	No
C6.2	**User Interface faults occur in Interaction modules**	**Yes**
C7.1	View modules have Framework faults	No
C7.2	**Framework faults occur in View modules**	**Weak**
C8.1	Error handling modules have Data faults	No
C8.2	Data faults occur in Error Handling modules	No
C9.1	Environ. Setup/Config. Modules have framework faults	No
C9.2	Framework faults occur in Environ. Modules	No

We conclude with some remarks about threats to validity of our case studies. As with any case study, there are unavoidable threats to validity. First, we cannot generalize the results to other application domains, systems, or languages. What we can say, however, is that we found support for our taxonomy in both the Apache and Mozilla systems. Second, a case study is limited in the amount of control over what data can be collected. We were limited by the available bug reports. While the random selection of defect reports for both systems does not bias the results, the quality and information content of the bug reports possibly could. Given the nature of case studies, we had no control over how bugs were reported. The analysis depends on the quality of the bug reports. They have to contain enough information for fault classification. While we have not performed a large scale inter-rater reliability analysis of the module classification, we used a team to classify them, in line with guidelines in [1]. The analysis is based on a theory about a fault link taxonomy that is based on existing knowledge and empirical studies as explained in section 4. It is thus possible that new fault links may be found, and, of course, some applications may not have certain faults. We consider our work a step towards building a more comprehensive theory.

7 Conclusions and Future Work

We have developed two taxonomies, one for modules and one for code faults. We introduced the notion of a fault link. We presented two methods for module classification along with their advantages and disadvantages. We classified modules and code faults of two open source, web-based software products using our approach.

We found evidence in favor of the existence of four conjectured fault links (and an additional two with weak evidence) and six fault links that were not conjectured (all related to Control/Logic faults). We have already capitalized upon the discovery of the Control/Logic fault links (for every module type) by augmenting our FTR checklists.

Table 7. Mozilla Fault Types

Fault	Number	Percentage
Data	13	17.3%
Computational	6	8%
Control/Logic	40	**53.4%**
Interface	6	8%
User interface	6	8%
Framework	4	5.3%
Total	75	100%

Table 8. Mozilla Module Types

Module Types	Number	Percentage
Data-centric	5	16.7%
Computational-centric	8	**26.7%**
Controller	6	20%
View	5	16.7%
Interaction	1	3.3%
Error Handling	1	3.3%
Environmental setup	4	13.3%
Total	30	100%

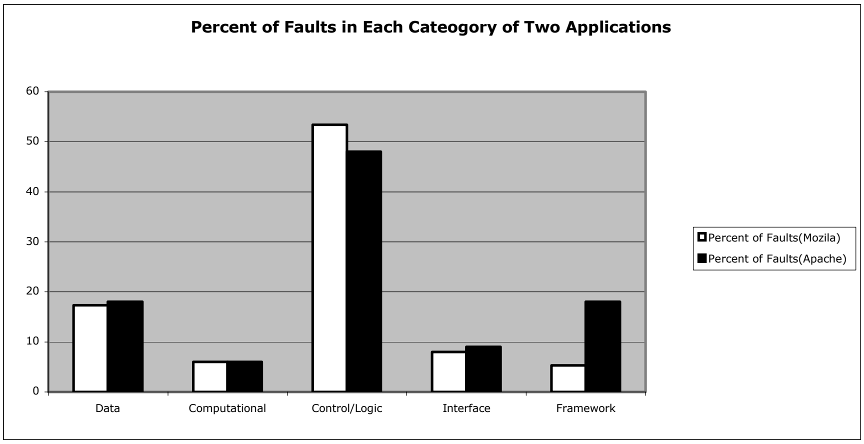


Fig. 2. Comparison of Fault Type Percentages

We continue work on the fault taxonomy and the module taxonomy and hope that others will assist us in validating and improving them. We plan to examine the taxonomies with respect to the object-oriented methodology. We plan to examine languages such as Lisp that provide control abstraction. We also are not convinced that the fault taxonomy is orthogonal. Specifically, we plan to evaluate mixed-purpose modules in the context of the fault link taxonomy. Our taxonomies might require tailoring to a specific domain or application, such as real-time or embedded systems, as discussed in [15]. We also plan to expand the fault link concept to fault chains. Faults rarely occur in isolation. They may be related longitudinally within a release (e.g., a design fault leads to a code fault) or across releases (e.g., incomplete fault repair). We refer to these relationships as fault chains. We have identified several types of fault chains, and will continue our work in this area. The ultimate goal of this work is to identify evaluation techniques that can take advantage of our knowledge of fault chains to prevent or detect faults as early as possible. That will assist us in developing reliable, though complex, software systems.

Acknowledgements

We thank Jeff Offutt for never tiring of discussions about testing and faults. Thanks to Martin Feather and Alan Nikora of JPL who looked at early versions of these taxonomies. Thanks to Kirk Kandt, also of JPL, for excellent comments on an earlier version of this paper.

References

1. Allen, M. and Yeh, W. *Introduction to Measurement Theory*. Brooks/Cole Publishing, 1979.
2. Apache modules and problem reports, Apache HTTP server version 1.3.24, http://httpd.apache.org/docs/mod/index-bytype.html.

3. Basili, V.R. and Barry T. Perricone. ‘‘Software Errors and Complexity: An Empirical Investigation.’’ *Communications of the ACM*, 27, 1 (January 1984), 42-51.
4. Beizer, B. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd Edition, ISBN 0-442-20672-0, 1990.
5. Bieman, J., Andrews, A. and H. Yang. Analysis of change-proneness in software using patterns: a case study, submitted *Seventh European Conference on Software Maintenance and Reengineering* (Benevento, Italy, March 2003).
6. Bieman,J., Jain, D., and H. Yang. Design patterns, design structure, and program changes: an industrial case study. *Proceedings of the International Conference on Software Maintenance* (Florence, Italy, 6 – 10 November 2001).
7. Centre of Software Maintenance, University of Durham, England. http://www.dur.ac.uk/ computer. science/research/csm/rip/introduction.html
8. Cooper, A. *About face: the essentials of user interface design.* IDG Books Worldwide, Foster City, CA, 1995.
9. Duncan, IMM., and Robson, DJ.: An exploratory study of common coding faults in C programs. A technical report, Centre for Software Maintenance, University of Durham, England, May 1991.
10. Endres, A. ‘‘An Analysis of Errors and Their Causes in System Programs’’. *Proceedings of the 1975 International Conference on Reliable Software*, in *SIGPLAN Notices*, vol. 10, No. 6, pp. 327-336, June, 1975.
11. Fenton, N.E., and Ohlsson, N. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering,* vol. 26, No. 8, August 2000, pp. 797-814.
12. Freimut, B. "Developing and Using Defect Classification Schemes", Fraunhofer IESE *IESE-Report No. 072.01/E, Version 1.0*, September, 2001.
13. Gall, H., Hajek, K., and M. Jazayeri. Detection of logical coupling based on product release history. *Procs. International Conference on Software Maintenance* (Bethesda, MD, November, 1998). IEEE Computer Society Press, 190-198.
14. Gram, C. A software engineering view of user interface design. Engineering for Human-Computer Interaction. *Proceedings of the IFIP TC2/WG2.7 working conference on engineering for human-computer interaction* (Yellowstone Park, USA, August 1995). Chapman & Hall, London, 1996, 293-304.
15. Hayes, J.H. “Building a Requirement Fault Taxonomy: Experiences from a NASA Verification and Validation Research Project,” *IEEE International Symposium on Software Reliability Engineering (ISSRE) 2003* (Denver, CO, November 2003).
16. Hayes, J.H., Dekhtyar, A., and J. Osbourne, “Improving Requirements Tracing via Information Retrieval,” in *Proceedings of the International Conference on Requirements Engineering* (Monterey, California, September 2003).
17. Hayes, J.H., Mohamed, N., and T. Gao, “The Observe-Mine-Adopt Model: An Agile Way to Enhance Software Maintainability”, *Journal of Software Maintenance and Evolution: Research and Practice*, *15, 5* (October 2003), 297 – 323.
18. IBM Research, Center for Software Engineering, "Details of ODC v5.11", http://www.research.ibm.com/softeng/ODC/DETODC.HTM.
19. IEEE Standard Classification for Software Anomalies, December 12, 1995. IEEE Std 1044.1-1995.
20. Lanubile, F., Shull, F., and V.R. Basili, “Experimenting with Error Abstraction in Requirements Documents”, *Proceedings of the 5th Inernational. Symposium on Software Metrics* (Bethesda, Maryland, 1998).

21. Macaulay, L. *Human -computer interaction for software designers.* International Thomson Computer Press, London, 1995.
22. Marick, B. A survey of software fault surveys. A technical report UIUCDCS-R-90-1651, University of Illinois, 1990; pp 2-23.
23. Mayhew, DJ. *Principles and guidelines in software user interface design.* Englewood Cliffs, N.J. Prentice Hall, 1992.
24. Mayrhauser, A., Ohlsson, MC., and Wohlin, C.**:** Deriving fault architecture from defect history. *J. Softw. Maint. Res. Pract., 12*, (2000), 287-304.
25. Miller, LA., Groundwater, EH., Hayes, J., and Mirsky, SM.**:** Guidelines for the verification and validation of expert system software and conventional software. SAIC 1995; 2**:** pp 100.
26. Mozilla organization website, http://mozilla.org/.
27. Munch, J, Rombach, H.D., Rus, I. Creating an advanced software engineering laboratory by combining empirical studies with process simulation. *Proceedings of the International Workshop on Software Process Simulation and Modeling (ProSim 2003)* (Portland, Oregon, USA, May 3-4, 2003).
28. Nikora, A., and Munson, J. Developing Fault Predictors for Evolving Software Systems. Proceedings of the Ninth International Software Metrics Symposium (METRICS 2003) (Sydney, Australia, September 2003).
29. Offutt, J. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering Methodology*, *1, 1* (January 1992), 3-18.
30. Offutt, J., and J. H. Hayes. A Semantic Model of Program Faults. *International Symposium on Software Testing and Analysis (ISSTA 96)* (San Diego, CA, January 1996).
31. Ohlsson, M., Andrews, A., and C. Wohlin. Modelling fault-proneness statistically over a sequence of releases: a case study. *Journal of Software Maintenance and Evolution: Research and Practice*, Volume 13, June 2001, pp. 167--199.
32. Ostrand, T. and Weyuker, W. The Distribution of Faults in a Large Industrial Software System. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA) 2002* and *ACM SIGSOFT*, vol. 27, No. 4, July 2002, pp. 55-64.
33. Perry, D.E., and C.S. Stieg, "Software Faults in Evolving a Large, Real-Time System: a Case Study", AT&T Bell Laboratories, *Proceedings of the 4th European Software Engineering Conference*, Garmisch, Germany, September 1993.
34. Rombach, H.D.., Basili, V., Selby, R. Experimental Software Engineering Issues: Critical Assessment and Future Directions. Lecture Notes in Computer Science. Springer Verlag, 1993.
35. Shneiderman, B. *Designing the user interface: strategies for effective human-computer interaction*. Addison-Wesley, Reading, MA, 1992.
36. Sullivan, M., and Chillarege, R. Software defects and their impact on system availability-A study of field failures in operating systems. *Digest 21st International Symposium on Fault-Tolerant Computing* (Montreal, Canada, June 1991).
37. Warren-Smith, RF.**:** Starlink project, Rutherford Appleton Laboratory, http://star-www.rl.ac.uk/star/docs/sgp42.htx/sgp42.html#stardoctoppage.
38. Wohlin, C. and Andrews, A. Analysing Primary and Lower Order Project Success Drivers. Proceedings of the Software Engineering and Knowledge Engineering (SEKE) 2002, Isclina, Italy, July 2002, CS Press.
39. Yu, WD., Barshefsky, A., and Huang, ST. An empirical study of software faults preventable at a personal level in a very large software development environment. Bell Labs Technical Journal 1997; 2**:** 221-232

Model-Based Identification of Fault-Prone Components*

Stefan Wagner and Jan Jürjens

Institut für Informatik,
Technische Universität München,
Boltzmannstr. 3, D-85748 Garching, Germany
`http://www4.in.tum.de/~{wagnerst, juerjens}`

Abstract. The validation and verification of software is typically a costly part of the development. A possibility to reduce costs is to concentrate these activities on the fault-prone components of the system. A classification approach is proposed that identifies these components based on detailed UML models. For this mainly existing code metrics are tailored to be applicable to models and are combined to a suite. Two industrial case studies confirm the ability of the approach to identify fault-prone components.

1 Introduction

The whole area of testing and quality assurance constitutes a significant part of the total development costs for software, often up to 50% [1]. Especially formal verification is frequently perceived as rather costly. Therefore there is a possibility for optimizing costs by concentrating on the fault-prone components and thereby exploiting the existing resources as efficiently as possible. Detailed design models offer the possibility to analyse the system early in the development life-cycle. One of the possibilities is to measure the complexity of the models to predict fault-proneness assuming that a high complexity leads to a high number of defects.

The complexity of software code has been studied to a large extent. It is often stated that complexity is related to and a good indicator for the fault-proneness of software [2, 3, 4]. There are two different approaches to the identification of fault-prone components. In the *estimative approach* models are used to predict the number of faults that are contained in each component. The *classification approach* categorizes components into fault-prone classes, often simply low-fault and high-fault. We use the latter approach in the following because it is more suitable for the model metrics.

* This research was supported in part by the *Deutsche Forschungsgemeinschaft (DFG)* within the project *InTime* and by the German Ministry for Science and Education within the *Verisoft* project.

Although the traditional complexity metrics are not directly applicable to design models because of different means of structuring and abstractions, there are already a number of approaches that propose design metrics, e.g. [5, 6, 7, 8]. Most of the metrics in [8] were found to be good estimators of fault-prone classes in [9] and are used in our approach as well. However, they concentrate mainly on the structure of the designs. Since the system structure is not sufficient as a source for the complexity of its components, which largely depends on their behavior, we will also propose a metric for behavioral models.

Contribution. This paper contains an adaption of complexity metrics to measure design complexity of UML 2.0 models. Based on these metrics an approach is proposed for deriving the fault-proneness of classes. Furthermore the metrics and the approach are validated by two industrial case studies.

Outline. In Sec. 2 complexity metrics for models built with a subset of UML 2.0 are defined and an approach for using the metrics to derive fault- and failure-prone components is explained. Two case studies are provided in Sec. 3. Finally, related work and conclusions are discussed in Sec. 4 and Sec. 5, respectively.

2 Analyzing Fault-Proneness

This section describes the possibilities to identify fault-prone components based on models built with UML 2.0 [10]. We introduce a design complexity metrics suite for a subset of model elements of the UML 2.0 and explain how to identify fault-prone components.

The basis of our metrics suite forms the suite from [8] for object-oriented code and the cyclomatic metric from [11]. In using a suite of metrics we follow [12, 13] stating that a single measure is usually inappropriate to measure complexity.

In [14] the correlation of metrics of design specifications and code metrics was analyzed. One of the main results was that the code metrics such as the cyclomatic complexity are strongly dependent on the level of refinement of the specification, i.e. the metric has a lower value the more abstract the specification is. Models of software can be based on various different abstractions, such as functional or temporal abstractions [15]. Depending on the abstractions chosen for the model, various aspects may be omitted, which may have an effect on the metrics. Therefore, it is prudent to consider a suite of metrics rather than a single metric when measuring design complexity to assess fault-proneness of system components.

Development Process. The metric suite described below is generally applicable in all kinds of development processes. It does not need specific phases or sequences of phases to work. However, we need detailed design models of the software to which we apply the metrics. This is most rewarding in the early phases as the models then can serve various purposes.

We adjust metrics to parts of UML 2.0 based on the design approach taken in AutoFocus [16], ROOM [17], or UML-RT [18], respectively. This means that we model the architecture of the software with structured classes (called actors in ROOM, capsules in UML-RT) that are connected by ports and connectors and which have associated state machines that describe their behavior.

The metrics defined in this section are applicable to components as well as classes. However, we will concentrate on structured classes following the usage of classes in ROOM. The particular usage should nevertheless be determined by the actual development process.

2.1 Measures of the Static Structure

We start introducing the new measures with the ones that analyze the static structure of models. These are important because the interrelations and dependencies among model elements contribute significantly to their complexity.

Structured Classes. The concept of structured classes introduces composite structures that represent a composition of run-time instances collaborating over communication links. This allows UML classes to have an internal structure consisting of other classes that are bound by connectors. Furthermore ports are used as a defined entry point to a class. A port can group various interfaces that are provided or required. A connection between two classes through ports can also be denoted by a connector. The parts of a class work together to achieve its behavior. A state machine can also be defined to describe behavior additional to the behavior provided by the parts.

We start with three metrics, Number of Parts, Number of Required Interfaces, and Number of Provided Interfaces, which concern structural aspects of a system model. The metrics consider composite structure diagrams of single classes with their parts, interfaces, connectors, and possibly state machines. A corresponding example is given in Fig. 1.

Number of Parts (NOP). The number of parts of a structured class contributes obviously to its structural complexity. The more parts it has, the more coordination is necessary and the more dependencies there are, all of which may contribute to a fault. Therefore, we define *NOP* as the number of direct parts C_p of a class.

Number of Required Interfaces (NRI). This metric is (together with the NPI metric below) a substitute for the old *Coupling Between Objects (CBO)* that was criticized in [19] in that it does not represent the concept of coupling appropriately. It reduces ambiguity by giving a clear direction of the coupling. We use the required interfaces of a class to represent the usage of other classes. This is another increase of complexity which may as well lead to a fault, for example if the interfaces are not correctly defined. Therefore we count the number of required interfaces I_r for this metric. Coupling metric as predictors of run-time failures were investigated in [20]. It shows that coupling metrics are suitable predictors of failures.

Number of Provided Interfaces (NPI). Very similar but not as important as NRI is the number of provided interfaces I_p. This is similarly a structural complexity measure that expresses the usage of a class by other entities in the system.

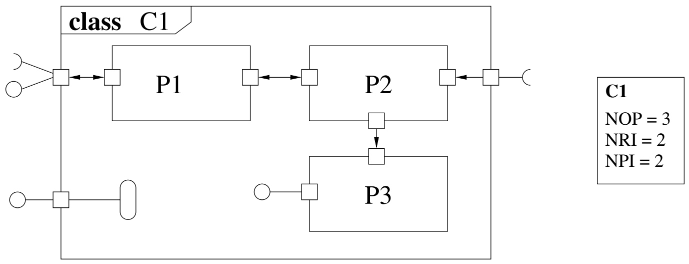


Fig. 1. An example structured class with three parts and the corresponding metrics

Example. The example in Figure 1 shows the composite structure diagram of a class with three ports, two required and two provided interfaces. It has three parts which have in turn ports, interfaces and connectors. However, these connecting elements are not counted in the metrics for the class itself because they are counted by the metrics for the parts, and these can later be summed up to consider the complexity of a class including its parts.

2.2 Measure of Behavior

We proceed with a complexity metric for behavioral models because the behavior determines the complexity of a component to a large extent.

State Machines. State machines are used to describe the behavior of classes of a system. They describe the actions and state changes based on a partitioning of the state space of the class. Therefore the associated state machine is also an indicator of the complexity of a class and hence its fault-proneness. State machines consist of states and transitions where states can be hierarchical. Transitions carry event triggers, guard conditions, and actions.

We use cyclomatic complexity [11] to measure the complexity of behavioral models represented as state machines because it fits most naturally to these models as well as to code. This makes the lifting of the concepts from code to model straightforward.

To find the cyclomatic complexity of a state machine we build a control flow graph similar to the one for a program in [11]. This is a digraph that represents the flow of control in a piece of software. For source code, a vertex is added for each statement in the program and arcs if there is a change in control, e.g. an if- or while-statement. This can be adjusted to state machines by considering the code implementation. The code transformation that we use as a basis for the metrics can be found in [17]. However, different implementation strategies could be used [21].

Example. An example of a state machine and its control flow graph is depicted in Fig. 2. At first we need an entry point as the first vertex. The second vertex starts the loop over the automata because we need to loop until the final state is reached or infinitely if there is no final state. The next vertices represent transi-

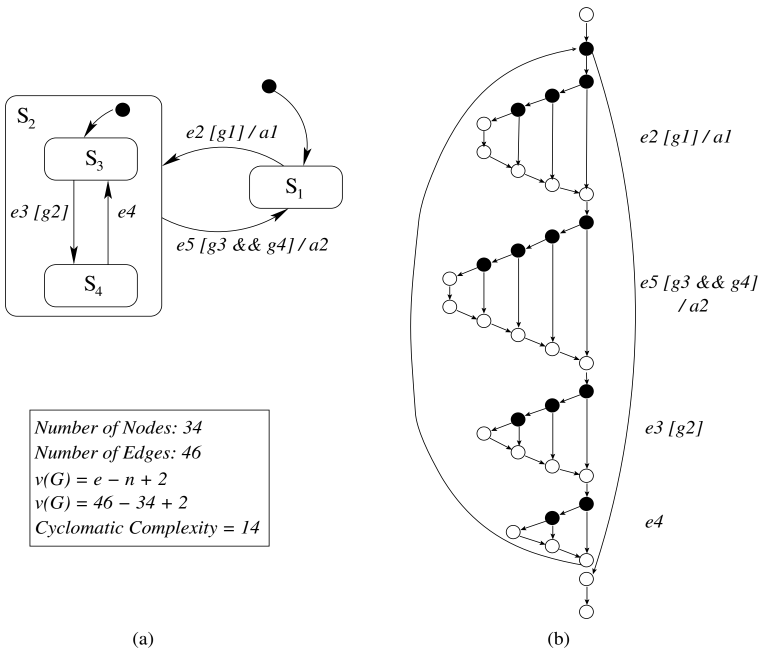


Fig. 2. (a) A simple state machine with one hierarchical state, event trigger, guard conditions, and actions. (b) Its corresponding control flow graph. The black vertices are predicate nodes. On the right the transitions for the respective part of the flowgraph are noted

tions, atomic expressions[1] of guard conditions, and event triggers of transitions. These vertices have two outgoing arcs each because of the two possibilities of the control flow, i.e. an evaluation to *true* or *false*. Such a branching flow is always joined in an additional vertex. The last vertex goes back to the loop vertex from the start and the loop vertex has an additional arc to one vertex at the end that represents the end of the loop. This vertex finally has an arc to the last vertex, the exit point.

If we have such a graph we can calculate the cyclomatic complexity using the formula $v(G) = e - n + 2$, where v is the complexity, G the control graph, e the number of arcs, and n the number of vertices (nodes). There is also an alternative formula, $v(G) = p + 1$, which can also be used, where p is the number of binary *predicate nodes*. Predicate nodes are vertices where the flow of control branches.

Hierarchical states in state machines are not incorporated in the metric. Therefore the state machine must be transformed into an equivalent state ma-

[1] A guard condition can consist of several boolean expressions that are connected by conjunctions and disjunctions. An atomic expression is an expression only using other logical operators such as equivalence. For a more thorough definition see [11].

chine with simple states. This appears to be preferable to handling hierarchy separately because we are not looking at understandability and we do not have to deal with hierarchy crossing transitions. Furthermore internal transitions are counted equally to normal transitions. Pseudo states are not counted themselves, but their triggers and guard conditions. Usage of the *InState* construct in guards is not considered.

Cyclomatic Complexity of State machine (CCS). Having explained the concepts based on the example flow graph above, the metric can be calculated directly from the state machine with a simplified complexity calculation. We count the atomic expressions and event triggers for each transition. Furthermore we need to add 1 for each transition because we have the implicit condition that the corresponding source state is active. This results in the formula

$$CCS = |T| + |E| + |A_G| + 2. \tag{1}$$

where T is the multi-set of transitions, E is the multi-set of event triggers, and A_G is the multi-set of atomic expressions in the guard conditions. This formula yields exactly the same results as the longer version above but has the advantage that it is easier to calculate.

For this metric we have to consider two abstraction layers. First, we transform the state machine into its code representation[2] and second use the control flow graph of the code representation to measure structural complexity. The first "abstraction" is needed to establish the relationship to the corresponding code complexity because it is a good indicator of the fault-proneness of a program. The proposition is that the state machine reflects the major complexity attributes of the code that implements it. The second abstraction to the control flow graph was established in [11] and is needed for the determination of paths through the program which reflect the complexity of the behavior.

2.3 Metrics Suite

In addition to the metrics which we defined above, we complete our metrics suite by adding two existing metrics from [8] that can be adjusted to be applicable to UML models. The metrics chosen are from the ones that were found to be good indicators of fault-prone classes in [9]. We omit *Response For a Class (RFC)* and *Coupling Between Objects (CBO)*[3] because they cannot be determined on the model level. The two adapted metrics are described in the following. The complete metrics suite can be found in Tab. 1.

Depth of Inheritance Tree (DIT). This is the maximum depth of the inheritance graph T to a class c. This can be determined in any class diagram that includes inheritance.

[2] Note that this is done only for measuring purposes; our approach also applies if the actual implementation is not automatically generated from the UML model but manually implemented.

[3] RFC counts all methods of a class and all methods recursively called by the methods. CBO counts all references of a class to methods or fields of other classes.

Number of Children (NOC). This is the number of direct descendants C_d in the inheritance graph. This can again be counted in a class diagram.

Table 1. A summary of the metrics suite with its calculation

Name	Abbr.	Calculation
Depth of Inheritance Tree	DIT	$max(depth(T, c))$
Number of Children	NOC	$\|C_d\|$
Number of Parts	NOP	$\|C_p\|$
Number of Required Interfaces	NRI	$\|I_r\|$
Number of Provided Interfaces	NPI	$\|I_p\|$
Cyclomatic Complexity of State machine	CCS	$\|T\| + \|E\| + \|A_G\| + 2$

We analyze whether our metrics are *structural complexity measures* by the definition in [12]. The definition says that for a set D of documents with a pre-order $\leq_D$ and the usual ordering $\leq_{\mathbb{R}}$ on the real numbers $\mathbb{R}$, a structural complexity measure is an order preserving function $m : (D, \leq_D) \longrightarrow (\mathbb{R}, \leq_{\mathbb{R}})$. This means that any structural complexity metric needs to be at least pre-ordered because this is necessary for comparing different documents. Each metric from the suite fulfills this definition with respect to a suitable pre-order on the relevant set of documents. The document set D under consideration is depending on the metric: either a class diagram that shows inheritance and possibly interfaces, a composite structure diagram showing parts and possibly interfaces, or a state machine diagram. All the metrics use specific model elements in these diagrams as a measure. Therefore there is a pre-order $\leq_D$ between the documents of each type based on the metrics: We define $d_1 \leq_D d_2$ for two diagrams d_1, d_2 in D if d_1 has fewer of the model elements specific to the metric under consideration than d_2. The mapping function m maps a diagram to its metric, which is the number of these elements. Hence m is order preserving and the metrics in the suite qualify as structural complexity measures.

Fault Proneness. As mentioned before, complexity metrics are good predictors for the reliability of components [2, 3]. Furthermore the experiments in [9] show that most metrics from [8] are good estimators of fault-proneness. We adopted DIT and NOC from these metrics unchanged, therefore this relationship still holds. The cyclomatic complexity is also a good indicator for reliability [2] and this concept is used for CCS to be able to keep this relationship. The remaining three metrics were modeled similarly to existing metrics. NOP resembles NOC, NRI and NPI are similar to CBO. NOC and CBO are estimators for fault-proneness, therefore it is expected that the new metrics behave accordingly.

The metrics suite is used to determine the most fault-prone classes in a system. Different metrics are important for different components. Therefore one cannot just take the sum over all metrics to find the most critical component. We propose to use the metrics so that we compute the metric values for each component and class and consider the ones that have the highest measures for each single metric. This way we can for example determine the components with complex behavior or coupling.

We suggest to use *complexity levels* $L_C = \{high, low\}$. We assign each component such a complexity level by looking at the extreme values in the metrics results. Each component that exhibits a high value in at least one of the metrics is considered of having the complexity level *high*, all other components have the level *low*. It depends on the actual distribution of values to determine what is to be considered a high value. These complexity levels show the high-fault and low-fault components.

Failure Proneness. The following constitutes an extension to the analysis of fault proneness towards failure proneness. The fault-proneness of a component does not directly imply low reliability because a high number of faults does not mean that there is a high number of failures [22]. However, a direct reliability measurement is in general not possible on the model level. Nevertheless, we can get close by analysing the failure-proneness of a component, i.e. the probability that a fault leads to a failure that occurs during software execution.

It is not possible to express the probability of failures with exact figures based on the design models. We propose therefore to use more coarse-grained *failure levels*, e.g. $L_F = \{high, medium, low\}$, where L_F is the set of failure levels. This allows an abstract assessment of the failure probability. It is still not reliability as generally defined but the best estimate that we can get in early phases.

To determine the failure level of a component we use the complexity levels from above. Having assigned these complexity levels to the components, we know which components are highly fault-prone. The operational profile [23] is a description of the usage of the system, showing which functions are mostly used. We use this information to assign *usage levels* L_U to the components. This can be of various granularity. An example would be $L_U = \{high, medium, low\}$. When we know the usage of each component we can analyze the probability that the faults in the component lead to a failure.

The combination of complexity level and usage level leads us to the *failure level* L_F of the component. It expresses the probability that the component fails during software execution. We describe the mapping of the complexity level and usage level to the failure level with the function *fp*:

$$fp = L_C \times L_U \longrightarrow L_F, \text{where } L_F = L_U \cup \{low\} \tag{2}$$

What the function does is simply to map all components with a high complexity level to its usage level and all component with a low complexity level to *low*. However, this is only one possibility how *fp* can look like.

$$fp(x, y) = \begin{cases} y & \text{if } x = high \\ low & \text{otherwise} \end{cases} \tag{3}$$

This means that a component with high fault-proneness has a failure probability that depends on its usage and a component with low fault-proneness has generally a low failure probability.

Having these failure levels for each component we can use that information to guide the verification efforts in the project, e.g. assign the most amount of inspection and testing on the components with a high failure level. Parts of critical systems such as an exception handler still need thorough testing although its failure level might be low. However, this is not part of this work.

3 Case Studies

This section presents two industrial case studies that use the classification approach based on the metrics suite and contains a discussion of the results and observations. Both case studies do not analyze the DIT and NOC metrics because the models do not contain inheritance.

3.1 Automatic Collision Notification

The first case study we used to validate our proposed fault-proneness analysis is an automatic collision notification system as used in cars to provide automatic emergency calls. First, the system is described and designed using UML, then we analyze the model and present the results.

Description. The case study was done in cooperation with a car manufacturer. The problem to be solved is that many accidents of automobiles only involve a single vehicle. Therefore it is possible that no or only a delayed emergency call is made. The chances for successful help for the casualties are significantly higher if an accurate call is made quickly. This has lead to the development of so called *Automatic Collision Notification (ACN)* systems, sometimes also called *mayday* systems. They automatically notify an emergency call response center when a crash occurs. In addition, manual notification using the location data from a GPS device can be made. We used the public specification from the *Enterprise* program [24, 25] as a basis for the design model. Details of the implementation technology are not available. In this case study, we concentrate on the built-in device of the car and ignore the obviously necessary infrastructure such as the call center.

Device Design. Following [24] we call the built-in device *MaydayDevice* and divide it into five components. The architecture is illustrated in Fig. 3 using a composite structure diagram of the device.

The device is a processing unit that is built into the vehicle and has the ability to communicate with an emergency call center using a mobile telephone connection and retrieving position data using a GPS device. The components that constitute the mayday device are:

– *ProcessorModule*: This is the central component of the device. It controls the other components, retrieves data from them and stores it if necessary.
– *AutomaticNotification*: This component is responsible for notifying a serious crash to the processor module. It gets notified itself if an airbag is activated.

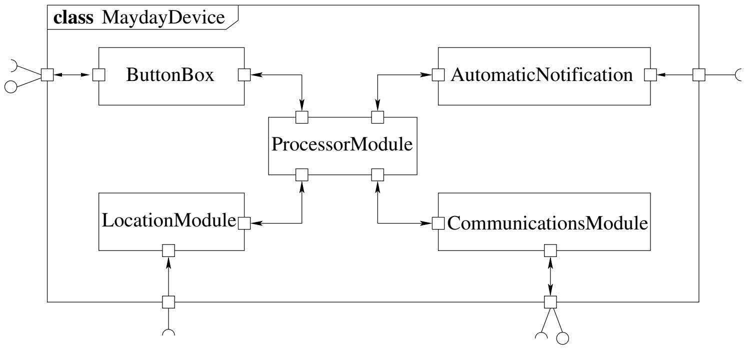


Fig. 3. The composite structure diagram of the mayday device

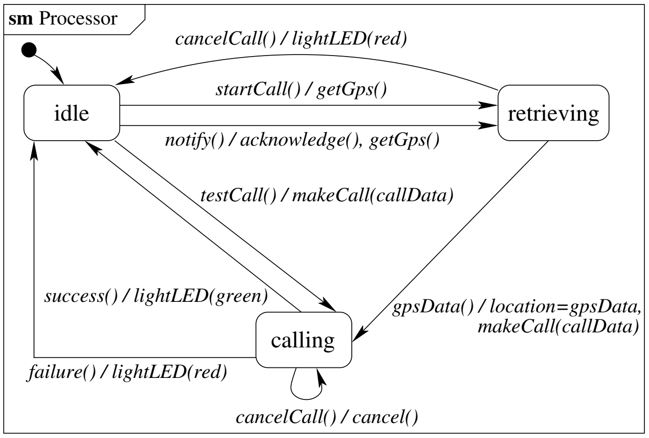

sm Processor
cancelCall() / lightLED(red)
idle
startCall() / getGps()
retrieving
notify() / acknowledge(), getGps()
testCall() / makeCall(callData)
success() / lightLED(green)
gpsData() / location=gpsData, makeCall(callData)
calling
failure() / lightLED(red)
cancelCall() / cancel()

Fig. 4. The state machine diagram of the *ProcessorModule*

– *LocationModule*: The processor module request the current position data from the location module that gathers the data from a GPS device.
– *CommunicationsModule*: The communications module is called from the processor module to send the location information to an emergency call center. It uses a mobile communications device and is responsible for automatic retry if a connection fails.
– *ButtonBox*: This is finally the user interface that can be used to manually initiate an emergency call. It also controls a display that provides feedback to the user.

Each of the components of the mayday device has an associated state machine to describe its behavior. We do not show all of the state machines because of space reasons but explain the two most interesting in more detail. This is, firstly, the state machine of the *ProcessorModule* called *Processor* in Fig. 4. It has three control states: *idle*, *retrieving*, and *calling*. The *idle* state is also the initial state. On request of an emergency call, either by *startCall* from the *ButtonBox* or *notify* from the *AutomaticNotification*, it changes to the *retrieving* state. This means that it waits for the GPS data. Having received this data, the state changes to *calling* because the *CommunicationsModule* is invoked to make the call. In case of success, it returns to the *idle* state and lights the green LED on the *ButtonBox*. Furthermore, the state machine can handle cancel requests and making a test call.

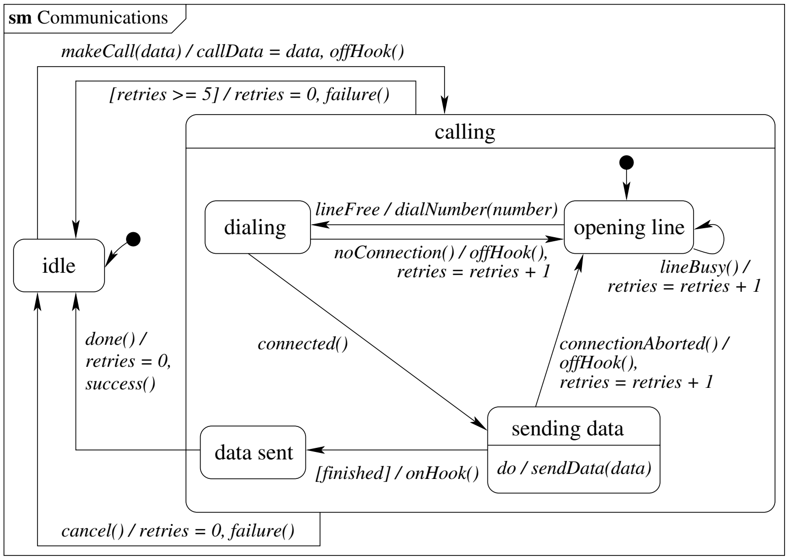

sm Communications
makeCall(data) / callData = data, offHook()
[retries >= 5] / retries = 0, failure()
calling
dialing
lineFree / dialNumber(number)
opening line
noConnection() / offHook(), retries = retries + 1
lineBusy() / retries = retries + 1
idle
done() / retries = 0, success()
connected()
connectionAborted() / offHook(), retries = retries + 1
sending data
do / sendData(data)
data sent
[finished] / onHook()
cancel() / retries = 0, failure()

Fig. 5. The state machine diagram of the *CommunicationsModule*

The second state machine is *Communications* in Fig. 5, the behavior specification of *CommunicationsModule*. One of the main complicating factors here is the handling of four automatic retries until a failure is reported. The state machine starts in an *idle* state and changes to the *calling* state after the invocation of *makeCall*. The *offHook* signal is sent to the mobile communications device. Inside the *calling* state, we start in the state *opening line*. If the line is free, the *dialing* state is reached by dialing the emergency number. After the *connected* signal is received, the state is changed to *sending data* and the emergency data is sent. After all data is sent, the *finished* flag is set which leads to the *data sent* state after the *onHook* signal was sent to the mobile. After the mobile sends the *done* signal, the state machine reports *success* and returns to the idle state. In case of problems, the state is changed to *opening line* and the retries counter is incremented. After four retries the guard [$retries\ >= 5$] evaluates to *true* and the call fails. It is also always possible to cancel the call which leads to a *failure* signal as well.

Results. The components of *MaydayDevice* are further analyzed in the following. At first we use our metrics suite from Sec. 2 to gather data about the model.

The results can be found in Tab. 2. It shows that we have no inheritance in the current abstraction level of our model and also that the considered classes have no parts apart from MaydayDevice itself. Therefore the metrics regarding these aspects are not helpful for this analysis.

Table 2. The results of the metrics suite for the components of *MaydayDevice*

Class	DIT	NOC	NOP	NRI	NPI	CCS
MaydayDevice	0	0	5	4	2	0
ProcessorModule	0	0	0	4	4	16
AutomaticNotification	0	0	0	2	1	4
LocationModule	0	0	0	1	2	4
CommunicationsModule	0	0	0	2	2	32
ButtonBox	0	0	0	2	2	8

More interesting are the metrics for the provided and required interfaces and their associated state machines. The class with the highest values for NRI and NPI is *ProcessorModule*. This shows that it has a high coupling and is therefore fault-prone. The same module has a high value for CCS but *Communications-Module* has a higher one and is also fault-prone.

In [25] there are detailed descriptions of acceptance and performance tests with the developed system. The system was tested by 14 volunteers. The usage of the system in the tests was mainly to provoke an emergency call by pressing the button on the button box.

The documentation in [25] shows that the main failures that occurred were failures in connecting to the call center (even when cellular strength was good), no voice connect to the call center, inability to clear the system after usage, and failures of the cancel function. These main failures can be attributed to the component *ProcessorModule* that is responsible for controlling the other components and *CommunicationsModule* that is responsible for the wireless communication. Therefore our analysis identified the correct components. The types of the corresponding faults of the failures are not available.

3.2 MOST NetworkMaster

We further validated our approach on the basis of the project results of an evaluation of model-based testing [26]. A network controller of an infotainment bus in the automotive domain, the MOST® NetworkMaster [27], was modeled with the case tool AutoFocus and test cases were generated from that model and compared with traditional tests. An implementation in C running on a standard PC was tested. We use all found faults from all test suites in the following. The AutoFocus notation is quite similar to UML 2.0 which allows straight-forward application of the metrics defined earlier.

Device Design. The composite structure diagram of the network master is shown in Figure 6. It contains two components *Divide* and *Merge* that are only responsible for the correct distribution of messages. The *MonitoringMgr* checks the status of devices in the network but has no behavior in the model, i.e. was functionally abstracted. The *RegistryMgr* is the main component. All devices need to register with it on startup and it manages this register. Finally, the *RequestMgr* answers requests about the addresses of other devices.

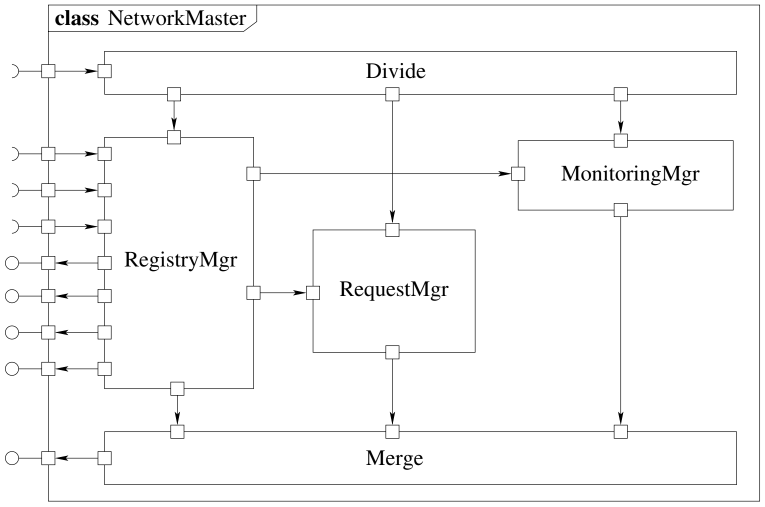


Fig. 6. The composite structure diagram of the MOST network master

We omit further parts of the design, especially the associated state machines, because of space and confidentiality reasons. The corresponding metrics are summarized in Table 3.

Table 3. The results of the metrics suite for the NetworkMaster

Class	DIT	NOC	NOP	NRI	NPI	CCS
NetworkMaster	0	0	5	4	4	0
Divide	0	0	0	1	3	11
Merge	0	0	0	3	1	8
MonitoringMgr	0	0	0	2	1	0
RequestMgr	0	0	0	2	1	14
RegistryMgr	0	0	0	4	7	197

Results. The data from the table shows that the *RegistryMgr* has the highest complexity in most of the metrics. Therefore we classify it as being highly fault-prone. As described in [26], several test suites were executed against an implementation of the network master. Some of which were developed manually, other based on existing Message Sequence Charts, and the remaining ones were automatically derived from an AutoFocus model. There were 24 faults identified by the test activities of which 13 are programming faults, 9 requirements defects, and 2 model faults. Of these faults 21 can be attributed to the *RegistryMgr* and 3 to the *RequestMgr*. There were no faults revealed in the other components. Hence, the high fault-proneness of the *RegistryMgr* did indeed result in a high number of faults revealed during testing.

3.3 Discussion

The two case studies confirmed our approach for identifying fault-prone components using model metrics. In both cases the suite ranked the components as high-fault that had code implementations which actually contained the most faults. Both models were developed completely independent of the implementations. Hence, model faults that lead to implementation faults cannot have an influence. Unfortunately, inheritance was not used in the studies. Therefore the validity of these metrics remains to be shown. It holds for the whole approach that the the external validity of the results of the case studies is limited as the small sample size does not allow a thorough statistical analysis.

Correlation of Metrics. A main problem of software metrics is that different metrics are often not independent. We analyse our proposed metrics suite concerning the correlation of the different metrics based on the data from the case studies. The sample size is small therefore the validity is limited but may give first indications.

We cannot analyse DIT and NOC because they were not used in the case studies. Also it does not make sense to analyse NOP with only two non-null data points. Therefore we concentrate on NRI, NPI, and CCS. The correlation between NRI / CCS and NPI / CCS is low with a a correlation coefficient $r = -0.17$ and $r = -0,13$, respectively. Only the correlation between NRI and NPI is more interesting. The correlation coefficient is 0.55 but the Chi-test and F-test only yielded probabilities of 0.35 and 0.17, respectively, for both data rows coming from the same population. Hence, we have a good indication that the metrics of our suite are not interdependent.

Correlation of Metrics and Faults. As we use the classification approach with our metrics, we cannot estimate numbers of faults and therefore a correlation between estimated and actual faults is not possible. Also a correlation analysis between the single metrics and the number of found faults is not helpful because only the combined suite can provide a complete picture of the complexity of the component. However, the statistical correlation between the metrics and the number of faults is not as low as expected. For NRI the coefficient is 0.35, for NPI 0.58, and for CCS 0.53 but chi- and f-tests showed a very low significance probably because of the small sample size.

Observations. By looking at the case studies it seems that the CCS metric has the most influence on the fault-proneness. However, there are components that do not have a state machine but their behavior is described by its parts and still might contain several faults. It also can be rather trivial to see that a specific component is fault-prone as in the case of the *RegistryMgr* of the *NetworkMaster*. This component has such a large state machine that it is obvious that it has to contain several faults. In larger models with a large number of components this might not be that obvious. Finally, there is no evident influence of the application type on the metrics visible from the case studies as both have components with a rather small number of interfaces and parts and a few components with quite large state machines.

4 Related Work

There have been few approaches that consider reliability metrics on the model level: In [7] an approach is proposed that includes a reliability model that is based only on the static software architecture. A complexity metric that is in principle applicable to models as well as to code is discussed in [5], but it also only involves static structure as well. In [6] the cyclomatic complexity is suggested for most aspects of a design metric but not further elaborated.

In et al. describe in [28] an automatic metrics counter for UML. They classify their metrics into various categories including fault proneness. The metrics in this category are WMC, NOC, and DIT. The latter two are the same as in our approach. The calculation of WMC is given as the sum of the complexities of the methods but no further explanation is given how this complexity should be calculated from the model. State machines and structured classes are not analysed.

A white paper from Douglass [29] contains numerous proposals of model metrics for all types of UML models. Therefore this work has several metrics that are not comparable to ours. Moreover, detailed explanations of the metrics is not available for all of them. Our DIT metric is similar to the *Class Inheritance Depth (CID)*, and NOC is comparable to *Number of Children (NC)*. The *Class Coupling (CC)* aims at a similar target as the NRI and NPI metrics but does not consider the interfaces but the associations. Finally, there is a complexity metric for state machines called *Douglass Cyclomatic Complexity (DCC)* that is also based on the metric from McCabe but handles nesting and and-states differently. Also triggers and guards are ignored. The whole intention of DCC is different to our CCS metric. Douglass considers more the aspect of the complexity in terms of comprehensibility whereas we want to capture the inherent complexity of the behavior of the component. Douglass gives rough guidelines for values that indicate "good" models but does not relate the metrics to fault proneness.

Other approaches have been used for dependability analysis based on UML models, although these do not consider complexity metrics: In [30] an approach to automatic dependability analysis using UML is explained where automatic transformations are defined for the generation of models to capture systems dependability attributes such as reliability. The transformation concentrates on structural UML views and aims to capture only the information relevant for dependability. Critical parts can be selected to avoid explosion of the state space. A method is presented in [31] in which design tools based on UML are augmented with validation and analysis techniques that provide useful information in the early phases of system design. Automatic transformations are defined for the generation of models to capture system behavioral properties, dependability and performance. There is a method for quantitative dependability analysis of systems modeled using UML statechart diagrams in [32]. The UML models are transformed to stochastic reward nets, which allows performance-related measures using available tools, while dependability analysis requires explicit modeling of erroneous states and faulty behavior.

5 Conclusions

We propose an approach to determine fault-prone components of a software system in the design phase already by complexity analysis of the design models. We use the concept of the cyclomatic complexity of code, lift it to the model level and combine it with adjusted object-oriented metrics originally from [8] to a metrics suite for UML 2.0. The metrics from [8] and [11] have undergone several experimental validations, e.g. [2, 9, 4, 3]. Because we used these metrics as a basis for our metrics suite we believe that it is a good indicator for fault-proneness. This was confirmed in two industrial case studies.

The metrics can also be used in conjunction with static analyses of the model concerning reliability [33].

For future work, we plan further experimental work to validate the approach. Furthermore, as soon as more data is available a discriminant analysis similar to [34] will be used to get a more solid mathematical foundation for the classification.

Acknowledgments

We gratefully acknowledge the joint work with M. Baumgartner, C. Kühnel, A. Pretschner, W. Prenninger, B. Sostawa, T. Stauner, and R. Zölch on the MOST Network Master. Furthermore we are grateful to M. Broy and W. Prenninger for commenting on a draft version and A. Pataricza for many helpful suggestions.

References

1. Myers, G.: The Art of Software Testing. John Wiley & Sons (1979)
2. Khoshgoftaar, T., Woodcock, T.: Predicting Software Development Errors Using Software Complexity Metrics. IEEE Journal on Selected Areas in Communications **8** (1990) 253–261

3. Munson, J., Khoshgoftaar, T.: Software Metrics for Reliability Assessment. In Lyu, M.R., ed.: Handbook of Software Reliability Engineering. IEEE Computer Society Press and McGraw-Hill (1996)
4. Rosenberg, L., Hammer, T., Shaw, J.: Software Metrics and Reliability. In: Proc. 9th International Symposium on Software Reliability Engineering (ISSRE'98), IEEE (1998)
5. Card, D., Agresti, W.: Measuring Software Design Complexity. The Journal of Systems and Software **8** (1988) 185–197
6. Blundell, J., Hines, M., Stach, J.: The Measurement of Software Design Quality. Annals of Software Engineering **4** (1997) 235–255
7. Wang, W.L., Wu, Y., Chen, M.H.: An Architecture-Based Software Reliability Model. In: Proc. Pacific Rim International Symposium on Dependable Computing (PRDC'99). (1999) 143–150
8. Chidamber, S., Kemerer, C.: A Metrics Suite for Object Oriented Design. IEEE Trans. Software Eng. **20** (1994) 476–493
9. Basili, V., Briand, L., Melo, W.: A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Trans. Software Eng. **22** (1996) 751–761
10. Object Management Group: UML 2.0 Superstructure Final Adopted specification (2003) OMG Document ptc/03-08-02.
11. McCabe, T.: A Complexity Measure. IEEE Trans. Software Eng. **5** (1976) 45–50
12. Melton, A., Gustafson, D., Bieman, J., Baker, A.: A Mathematical Perspective for Software Measures Research. IEE/BCS Software Engineering Journal **5** (1990) 246–254
13. Fenton, N., Pfleeger, S.: Software Metrics. A Rigorous & Practical Approach. 2nd edn. International Thomson Publishing (1997)
14. Henry, S., Selig, C.: Predicting Source-Code Complexity at the Design Stage. IEEE Software **7** (1990) 36–44
15. Prenninger, W., Pretschner, A.: Abstractions for Model-Based Testing. In Pezze, M., ed.: Proc. Test and Analysis of Component-based Systems (TACoS'04). (2004)
16. Huber, F., Schätz, B., Schmidt, A., Spies, K.: AutoFocus: A tool for distributed systems specification. In Jonsson, B., Parrow, J., eds.: Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium, FTRTFT'96. Volume 1135 of LNCS., Uppsala, Sweden, Springer (1996) 467–470
17. Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. John Wiley & Sons (1994)
18. Selic, B., Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. Available at http://www-106.ibm.com/developerworks/rational/library/ (1998)
19. Mayer, T., Hall, T.: A Critical Analysis of Current OO Design Metrics. Software Quality Journal **8** (1999) 97–110
20. Binkley, A., Schach, S.: Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures. In: Proc. 20th International Conference on Software Engineering (ICSE'98), IEEE Computer Society (1998) 452–455
21. Pintér, G., Majzik, I.: Program Code Generation based on UML Statechart Models. In: Proc. 10th PhD Mini-Symposium, Budapest University of Technology and Economics, Department of Measurement and Information Systems (2003)
22. Wagner, S.: Efficiency Analysis of Defect-Detection Techniques. Technical Report TUMI-0413, Institut für Informatik, Technische Universität München (2004)
23. Musa, J.: Software Reliability Engineering. McGraw-Hill (1999)
24. The ENTERPRISE Program: Mayday: System Specifications (1997) Available at http://enterprise.prog.org/completed/ftp/mayday-spe.pdf (January 2005).

25. The ENTERPRISE Program: Colorado Mayday Final Report (1998) Available at http://enterprise.prog.org/completed/ftp/maydayreport.pdf (January 2005).
26. Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., Stauner, T.: One Evaluation of Model-Based Testing and its Automation. In: Proc. 27th International Conference on Software Engineering (ICSE'05). (2005) To appear.
27. MOST Cooperation: MOST Media Oriented System Transport—Multimedia and Control Networking Technology. MOST Specification Rev. 2.3. (2004)
28. In, P., Kim, S., Barry, M.: UML-based Object-Oriented Metrics for Architecture Complexity Analysis. In: Proc. Ground System Architectures Workshop (GSAW'03), The Aerospace Corporation (2003)
29. Douglass, B.: Computing Model Complexity. Available at http://www.ilogix.com/whitepapers/whitepapers.cfm (January 2005) (2004)
30. Bondavalli, A., Mura, I., Majzik, I.: Automated Dependability Analysis of UML Designs. In: Proc. Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. (1999)
31. Bondavalli, A., Dal Cin, M., Latella, D., Majzik, I., Pataricza, A., Savoia, G.: Dependability Analysis in the Early Phases of UML Based System Design. Journal of Computer Systems Science and Engineering **16** (2001) 265–275
32. Huszerl, G., Majzik, I., Pataricza, A., Kosmidis, K., Cin, M.D.: Quantitative Analysis of UML Statechart Models of Dependable Systems. The Computer Journal **45** (2002) 260–277
33. Jürjens, J., Wagner, S.: Component-based Development of Dependable Systems with UML. In Atkinson, C., Bunse, C., Gross, H.G., Peper, C., eds.: Embedded System Development with Components. Springer (2005) To appear.
34. Munson, J., Khoshgoftaar, T.: The Detection of Fault-Prone Programs. IEEE Trans. Software Eng. **18** (1992) 423–433

Regression Test Selection for Testable Classes

Eliane Martins and Vanessa Gindri Vieira

Institute of Computing – Unicamp, Av. Albert Einstein 1251,
Campinas, ZC: 13083-970, SP, Brazil
{eliane, ra000463}@ic.unicamp.br

Abstract. A reusable class must be tested many times: each time modifications are applied to it or its base classes; when a subclass is created, in which case the inherited and redefined features must be retested in the new context. Therefore, a class should be easy to test, specifically for test execution and results analysis, since these activities must be repeated often. Inspired by R. Binder's self-testing class concept [4] we defined, in a previous work, a testable class as a 3-tuple: class implementation, class behavior model and built-in test (BIT) mechanisms. In this work we present how to use this information when a class is changed. Regression testing is necessary each time a software is changed, to assure that the modifications do not adversely affect the unchanged parts. It is assumed that the test suite applied when testing the old version is available for reuse. However, test suites can be large and require too much effort to be reapplied in their totality. In such cases, a subset of the tests must be selected. This selection usually requires extra information besides the source code. This work aims at answering the following question: how to use test information contained in a testable class to do regression testing? The answer involves, among other aspects, the definition of an approach to select tests for reuse. The approach can be fully automated and does not need the source code for regression-test selection.

1 Introduction

Regression testing is applied to modified software to make sure that modified parts behave as intended and do not propagate unintended side effects to unmodified parts of the software [21].

Regression testing is required each time a software changes, which can occur in several situations [5, ch.15.1.2]: when a bug fix has been completed, when a new increment is generated and has to be integrated with previous increments, when a new system build is generated, just to mention a few.

With the widespread use of object-oriented development and the emphasis on code reuse, the need for regression testing increases. For example, when a new subclass is created, the base class tests must be rerun on the new subclass. Also, when a base class is changed, it must be retested as well as all its subclasses.

Regression testing assumes the existence of the test suite applied to the old version. Reuse of existing test suites can reduce regression testing efforts, but these suites can be large, and one may not have enough time to rerun all tests in such suites. In this case, efforts must be restricted to a subset of the existing tests. The problem of choosing an appropriate subset of an existing test suite is called the *selective retest problem* and the techniques for solving this problem, *selective retest techniques* [21].

In this paper we present a selective retest technique to be applied when a class is modified. Our technique uses a specification-based approach, based on a class behavior model, to select tests in an existing test suite. Besides, the method is based on information added to a class to improve its testability. Software testability encompasses all aspects that ease software testing, from the quality of its specification, design, code, and tests, to the availability of test support [4]. Since a reusable class must be retested each time it is modified or reused in a new context, test activities, in special, test execution and results analysis should not require too much effort. In a previous work we have proposed an approach for the construction of a testable class [17]. This approach was based on the concept of self-testing class [4], in which a class is augmented with built-in test (BIT) capabilities and a test specification. BIT capabilities include the means to observe an object's state as well as monitor intermediate results by the use of assertions. The test specification, in our approach, is a behavior model for the objects of the class. This model is used for test case generation purposes, and in this paper we show how it can also be used for selective regression testing.

Therefore, this paper addresses the regression test selection (RTS) problem at the class level. In other words, we want to answer the following question: given a testable class C which has been tested with a test suite T, how can we use its additional information to build a test suite T', a subset of T, which can reveal faults in a modified version of C?

To answer this question we have to consider several problems. First, we need to adapt existing RTS techniques to incorporate the use of the testable class information. More precisely, given that we have a class behavior model, we need a specification-based selective regression test method. Second, we need information about which parts of the model are covered by each test case of T. Finally, we need information about the changes made to the class.

The main contributions of this study are the following: (i) regression test selection based on a behavioral model instead of a structural model (based on source code), which is useful when testing black-box components; (ii) mapping of an UML Activity Diagram to the model used for regression test analysis; (iii) use of design for testability information added to a class to guide regression testing.

In the next section we provide background information about regression testing and existing selection techniques. Section 3 addresses testability issues and shows our approach to build a testable class. In Section 4 we briefly describe the test model. Section 5 contains a description of the specification-based selective testing technique at the class level. Section 6 presents results of the application on a case study and Section 7 concludes the paper.

2 Regression Testing

2.1 Terminology

As already stated, regression testing is the re-execution of tests that have already been applied to an old version of a program P. A regression test case is a test case that P has passed (i.e., for which P terminated successfully), and which is expected to pass when is rerun on a modified version P'. A test case that causes P' to behave differently than P, reveals the presence of faults, designated as regression faults. A regression test suite is composed by regression test cases [5, ch.15.1].

One regression testing strategy consists of reruning all tests in a regression test suite (from now on referred to as test suite), in a so called retest-all strategy. Another strategy consists of selecting a subset of the test suite for re-execution. Based on information stored with the test, together with information about the changes, relevant test cases are identified [16]. This is designated as selective strategy. Both strategies perform the same activities, namely, test re-execution, results analysis and test generation for the parts added to the software. However, the retest-all strategy does not do any analysis before re-running the tests. If the effort to select a subset of tests to run is less than the effort to execute and analyze results of the tests that are omitted, the selective regression testing is economically more interesting than the retest-all strategy [16].

Regression testing is typically accomplished in two phases [21]: preliminary and critical. In the preliminary phase, programmers enhance and/or correct the software; in the meantime, test analysts can be developing test plans for the modified parts. When modifications are completed, the critical phase begins. Efforts to reduce regression testing costs aim at confining expensive activities on the preliminary phase.

2.2 Selective Regression Techniques

When using a selective regression technique, one wishes that the reduced test suite does not omit test cases that can reveal a regression fault. A regression test suite is safe when it consists of all test cases that can reveal faults in the modified component (or system).

A number of selective regression testing techniques have been proposed in the literature. Most of them are based on source code control flow or data flow analysis [11, 12, 22, 23]. Based on information about code modification of the two versions of a program (or class), as well as the parts of the program exercised by the tests when running on the old version, test cases are selected. These techniques can be safer and more precise in selecting test cases that cover the modifications than those based on requirements or design changes, as ours. A drawback is that they require that the changes be already implemented, which means that much of the regression testing effort is left to the critical phase. There are a few techniques that use specification information. One of these works presents an RTS technique based on an Activity Diagram [7], just like ours. Their work, however, do not address class testing specifically. Briand et al present an RTS technique based on OO design that uses UML class and case/sequence diagrams [6]. Our work also considers design changes as theirs, but uses the Activity Diagram instead. Besides, their work considers more than one class whereas ours considers a single class. None of the aforementioned works consider the use of design for testability information to help in regression testing as ours does.

3 Testable Class

Design for testability techniques and the self-testing concept have been used in hardware for a long time; however, only recently this subject has earned more attention in the software community.

As pointed out by different authors [4, 9], a testable software might possess a set of attributes such as observability, controllability and understandability. Roughly speaking, the more we can observe the states and outputs of a component, the more we can control it during testing; and the more information we have about a component, the easier it is to effectively test it.

A pioneering work in software testability is the approach proposed by D. Hoffman, which adds a test interface to a module in order to achieve the necessary control and observation during test execution [13]. Integrating assertions in the source code is also proposed in his work. An assertion is a Boolean expression that defines necessary conditions for correct execution of the software [5, ch.17]. Besides, he proposes that test suites be described in a specific notation from which drivers[1] are automatically generated.

R. Binder adapts this approach to the OO context, proposing the construction of **self-testing classes** [4]. A self-testing class is composed by the class (or component) under test (CUT) augmented with built-in test (BIT) capabilities and a test specification. On one hand, with the BIT capabilities it is possible to access, control and observe an object's internal state as well as monitor intermediate results by the use of assertions. On the other hand, the test specification is a detailed description about how to execute the tests cases, and can be used for test generation and also as a test oracle.

Several approaches have been proposed since then. In some of them, test cases generated during development are delivered with the component [4, 14, 24]. An advantage of these approaches is that effort is reduced in the re-execution of those test cases.

Other approaches can augment the component with information that component users might require for analysis and testing tasks. In the Self-Testing COTS Component (STECC) strategy, for example, components are augmented with analysis and testing functions [3]. Another approach proposes the use of a component metacontent, which consists of information (metadata) about components and utilities (metamethods) for computing and retrieving such information [19]. Metacontents are not restricted to testing information; instead, they can support some of software engineering tasks that depend on and can benefit from information about external components.

[1] A driver is a class, main program or external software that applies test cases to the software under test [4, ch.13.1].

In a previous work we proposed an approach inspired by the self-testing class concept mentioned above to build a testable class [17]. Differently from that approach, a test model is used instead of a test specification, representing the class objects behavior. A tool retrieves this test model for test case generation purposes. Since this model is the core of the RTS technique we propose, it is described in more detail in Section 4.

The embedded test information or built-in test (BIT) capabilities comprises: assertions, a reporter method and a BIT access control. The assertions implement the class contract, according to the Design by Contract concept [18]. A contract describes the relationship between an object's interface and a client. This contract can be guaranteed by implementing certain runtime checks, which are the preconditions, post-conditions and invariants. A precondition checks whether the object's clients have fulfilled their part of the contract. This usually means that input parameters are checked before a method is executed, to make sure that these parameters are appropriate for use in the method. A post-condition checks the results of the processing performed by the method. It is checked at the end of the method, just before its return. An invariant checks whether the state of the object is maintained between method calls. It is usually checked just after an object's construction, upon entry to a method and before leaving the method. In the C++ version used in this study, macros were used to implement these checks. Except for invariant checking which is performed by the Invariant( ) method because this checking can be very complex. Therefore, the testable class interface comprises the Invariant( ) and Report ( ) methods, the latter being responsible for reporting an object's state.

The BIT features can only be accessed if the class is in test mode, which is set by the user through BIT access control capability. This control capability prevents the misuse of BIT services and for the moment it consists of a compiler directive, which includes or excludes these capabilities. More details about our approach for constructing and using a testable class can be found on [17].

4 The Class Behavioral Model

The model used to represent the class behavior shows the sequential constraints (or precedence relation) among the messages the class can accept.This model is simpler than a state transition or a sequence diagram, in which it specifies what can be done, and in what order, but it is neither concerned with object states nor with the events that cause them to change. Our purpose was to have a simple model, quite similar to the control-flow graph model that represents the code structure of a program. The reason is that this model was the one used by Rothermel and Harrold regression test selection algorithm [20] which is used in this study.

The UML Activity Diagram (AD) fits our purposes, but some constraints must be made about the way it can be used. First of all, each activity represents processing of a message that can be sent to a class object. A transition from A to B in this diagram indicates that A must be sent before B, but sending A does not imply the sending of B. Second, we use a subset of the AD, as illustrated in Figure 1. This subset is based on an UML version 1.3 described in [8, ch.9]. The descriptions given below are based on this reference.

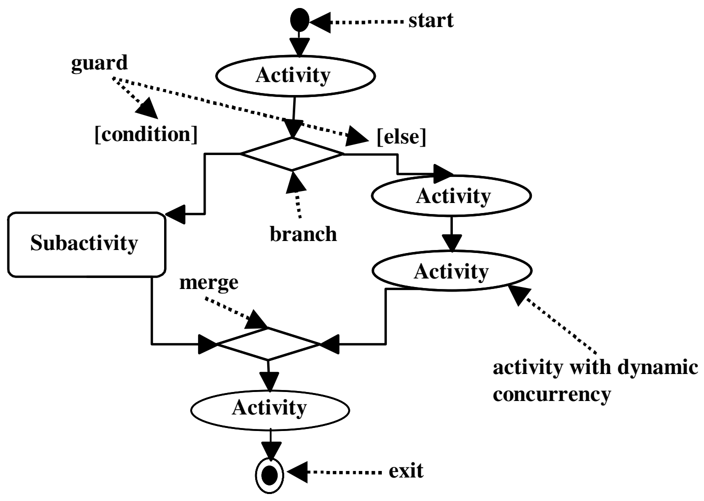


Fig. 1. Elements of the Activity Diagram considered in this study

The AD can support conditional behavior, which is delineated by branches and merges. A branch is a single incoming transition and several guarded outgoing transitions. A guard is a condition that must be true in order to traverse a transition. Only one of the outgoing edges can be taken, so the guards must be mutually exclusive. Using "[else]" as a guard in a transition indicates that this transition should be taken only if all other guards from the branch are false. A merge marks the end of a conditional behavior started by a branch. It has various incoming transitions and only one outgoing. Branches and merges can be represented using decision points (the diamonds shown in the figure).

An AD may be divided into sub-activities which allow the hierarchical organization of the diagram. Sub-activities may be detailed in a separate diagram for the sake of readability.

An AD also allows dynamic concurrency, which is useful to represent interactions without the use of loops in the diagram. A multiplicity mark (*) is used to indicate that a (sub-) activity can be executed many times.

It is also worth noting that a class being modeled may interact with other entities (for instance human beings, components, other objects) to accomplish a task. For the purpose of this study, all external references (databases, files, methods and others) are considered as part of a method's internal processing and, for those reasons, do not appear in the diagrams. Of course, stubs[2] may have to be constructed for testing purposes but this is beyond the scope of this text.

Activity Diagrams can also represent parallel behavior, in which various activities can be executed concurrently. This means that for these activities, the precedence relation does not hold, i.e., they can be executed in any order. For that reason, concurrent behavior is not being considered in this study. One way to overcome this

[2] A stub is a partial, temporary implementation of a component, serving as a placeholder for the parts of the software that are not tested but are used by the software under test.

situation is to encapsulate concurrent activities in sub-activities, the latter being executed according to the precedence relation in the diagram containing it.

Swim lanes, which allow the organization of activities among a group, are also not considered because our focus is on a single class.

5 Selective Regression Test Approach

Our RTS analysis is accomplished according to the following steps: (i) trace test cases to the elements of the behavioral model that are covered during the old version testing; (ii) perform change analysis to identify the modifications between the old and new versions of the behavioral model; (iii) identify the model elements affected by the changes and (iv) select test cases from the original test suite to reapply on the new version. These steps are detailed in the rest of this section, but first we formalize the behavioral model and introduce some notation and terminology necessary to explain the analysis approach.

5.1 The Test Model

A test model, which is a directed graph, is derived from the AD constructed according to what is described in Section 4, in the following way: each vertex in the graph corresponds to an activity or sub-activity, as well as the start and exit marks of the AD. The edges correspond to transitions in the AD, including branches and merges. In this way we obtain a control flow graph that represents the behavior instead of the source code structure of a class. It is designated as behavioral control flow graph(BCFG) [2, ch.3].

We now formalize this model, presenting the terminology and notation that will be used in the sequel. Most of the notation used here is based on [1].

The class BCFG is represented by a graph, $G= (V, E, s, x)$ with vertices V, edges E, a unique entry vertex s and an exit vertex x. We assume that all vertices are reachable from s, and x is reachable from all vertices. Each vertex has a label, which is the signature of an interface method, containing method name, arguments' names and types, and return value type. A sub-activity is considered as a method without parameters. Entry vertex s has the label "start" and exit vertex has the label "exit". The notation $S(v)$ designates the label of a vertex v.

Edges are also labeled. Labels are in the form of [condition] or [else], according to the corresponding guard in the AD. If a vertex has only one outgoing edge, its label is "ε", not shown in the figures to avoid cluttering them. An edge is designated as $e = (v, w, l)$, where v is the source vertex, w is the target vertex and l is the edge label. Labels uniquely identify the outgoing edges of a vertex.

A path in G is a sequence of edges: $p = [e_1, \ldots, e_n]$, where the target vertex of e_i is the source vertex of e_{i+1} for $1 \leq i \leq n$. A path can also be represented as a sequence of vertices and edge labels: $p = [v_1, l_1, \ldots, l_n, v_{n+1}]$, The ith vertex of p is denoted $p_v[i]$, and the ith edge label, $p_l[i]$.

A path from s to x is called a complete path. In a class BCFG, it represents a possible life history of an object, from creation to destruction. In this study, since the graph is acyclic, paths are loop-free, i.e., no edge is repeated in the sequence that constitutes the path.

A test case traverses (or covers) a complete path in G. Since G can be considered as an executable model, we refer to the execution of a test case t on G, denoted by G(t). It is worth noting that differently from a structural control flow graph, which is obtained from the program source code, a path in a BCFG may not correspond to a path in the implementation, since this one is considered as a black-box. Hence, unless explicitly stated, each time we refer to a path traversed by a test case we mean a path in the model, not in the implementation.

Various test cases can traverse the same path p in G with different test data or having a different configuration. Test case design is not our concern in this paper, as this was the subject of another study [17]. The focus here is on regression testing, so we assume that test cases are already available for reuse. They may have been generated using the test strategy described in the aforementioned reference or not. What interests us here is to determine the paths traversed by each test case, as it will be shown in the next section.

5.2 Tracing Test Cases to the Behavioral Model

Every test case design method implies a coverage metric that measures the extent to which the elements required by the test method have been exercised by a given test suite [5, ch.3.2]. For example, for a test method based on a control-flow graph, coverage metrics can be the percentage of vertices covered or the percentage of edges covered at least once by a test suite.

There are several tools that support control-flow coverage metric for source code. Basically, they add instrumentation statements to the code that then will be executed and the coverage metrics will be recorded. In this study, where the behavioral model is of concern rather than the source code, to measure coverage we need to implement the model. As suggested by B. Beizer, the model can be programmed in a language such as C/C++, Java, or any other supported by a coverage tool [2, ch.3.5]. As this author points out, implementing the model and the real software are distinct things. The model implementation does not include details such as I/O, operating system interfaces or anything else. Also, it does not need to run on the target platform, efficiency is not a concern either and finally, it does not have to be integrated with the rest of the software.

Having stated that, let us consider the path p covered by a test case t when running on G, the behavioral model of the old version of a class C. Since the elements of interest to the control-flow analysis are the edges, we obtain the edges covered by each test case when running on G. We denote this set of covered edges by E_p.

5.3 Change Analysis

In this step one has to consider the changes to be applied to the class and how to map them to the BCFG.

Modifications to a class can be considered as either contract changes or implementation changes [5, ch.15.2]. A contract change is a consequence of requirements or design changes that affect the class external interfaces. Examples of such a modification are: addition or removal of interface methods, alteration of method signature or changes that alter the external visible contract of the class, that is, alteration of pre- and post-condition methods, or class invariant.

An implementation change comprises all changes to a class that are not visible to its clients. In other words, this kind of change does not alter the class external behavior.

The class behavior model must be updated to reflect the changes to a class. Some changes affect the control-flow, causing addition or removal of vertices or edges. It is worth noting that a change in a vertex or edge label is considered as a removal of old vertex/edge and an addition of new ones. However, other changes do not affect the BCFG. For example, if a method maintains the same signature but its post-condition changes, or when it accesses an attribute that has changed, the BCFG, built as explained in section 5.1, is not affected. Since our analysis is based on this model, we need some way to identify the vertices whose changes are not "visible" to the BCFG.

Let $v \in V$ designate a vertex in the old version of the graph G with signature $S(v)$. Let $v' \in V'$ designate a vertex in the new version, and in the new version G' with signature $S(v')$. We define a set $V^c \subseteq V$ as follows: if v and v' are such that $S(v) = S(v')$ but v and v' differ in some way, then $v \in V^c$.

It is worth noting that V^c comprises any changes, including implementation changes. However, the latter are not the focus of this study because we are interested in changes that can be determined before the implementation phase. This does not mean that these changes do not deserve regression testing. Of course, the analysis presented here can be used in this case too, but the only point is that this analysis must be postponed to the implementation phase, which can have an impact on regression test schedules.

5.4 Identification of the Affected Edges

To identify the edges affected by the changes we use the graph walk algorithm as presented in [1, 20]. Some assumptions must be established in order to apply this algorithm. The first one, which is a concern of our approach, is that G' is an updated version of G, according to the changes made to the class. Second, which is also our concern, the set V^c is also defined.

A third assumption is that every complete path through G is potentially executable [1]. A path p in G is executable when it is possible to find input parameter values that will cause p to be traversed when running t on G. As mentioned earlier, we consider the path to be executable on the model, and not in the implementation. Although this is highly desirable, it is in practice very difficult to verify in black-box testing. One way to determine whether a behavioral control flow path is executed is to check intermediate processing results [2, ch.3]. In this way, the BIT capabilities (c.f. Section 3) can be very helpful.

Before defining the fourth assumption, let us first introduce some definitions. The graph walk algorithm mentioned usually determines the equivalence between two vertices of G and G'. Like in the other studies based on this algorithm, we use textual equivalence, i.e., two vertices v and v' are equivalent if their signatures, $S(v)$ and $S(v')$ are lexicographically identical. Let $Equiv(v, v')$ be true if v is equivalent to v'. Using this definition, one can also establish the equivalence between two paths. The paths p and q are identical if [1]: (i) p and q have the same length (same number of edges); (ii) Equiv ($p_v[i]$, $q_v[i]$) for all i, and (iii) $p_l[i] = q_l[i]$) for all i.

```
Begin
$D_e = \emptyset$
$R_e = \emptyset$
$V_{vis} = \emptyset$      // set of visited vertices
Classify_edges (s, s')
End

Procedure Classify_edges (v, v')
Begin
if $v \notin V_{vis}$ then
    $V_{vis} = V_{vis} \cup \{v\}$
    foreach edge $e = (v, w, l) \in E$ do
        w' = succ(v', l)
        if w' is null then     // the edge has been removed
            $D_e = D_e \cup \{e\}$
        else
            if Equiv(w, w') then
                if $w \neq x$ then
                    if $w \in V^c$ then      // w changed but S(w) is unchanged
                            $R_e = R_e \cup \{e\}$
                    fi
                    Classify_edge (w, w')
                fi
            else          // S(w) has changed or w has been removed
                    $D_e = D_e \cup \{e\}$
            fi
        fi
    od
fi
end
```

Fig. 2. Algorithm that obtains affected edges

Then we can define the fourth assumption, also called Controlled Regression Testing Assumption [20], which states the following: if G(t) traverses complete path p and G' contains a complete path p' equivalent to p, then G'(t) will traverse p' and have the same observable behavior as G(t).

For this assumption to hold, it must be ensured that regression testing for C' occurs in the same context (operating system, runtime system, compiler, etc) used for testing C. This is not a restrictive assumption either because this is common use in testing, in case of failure occurrence, so that the same behavior can be reproduced [12].

Under these four assumptions, the algorithm presented in Fig. 2 is applied to G and G' to obtain the affected edges. As the graph walk algorithm from Rothermel and Harrold [20], it consists of a synchronous depth-first search of the two graphs, with some modifications. Instead of having only one set containing the affected edges, we have two sets: D_e (set of deleted edges) and R_e (set of retestable edges). As it will be shown in the next section, these sets will drive test case selections.

The algorithm is always called with equivalent vertices v and v'. If v has already been visited, the algorithm returns. Otherwise, v is inserted in the set of visited vertices, V_{vis}, and each of its outgoing edges, e, are considered in turn. The function succ(v, l) returns the vertex w which is the successor of v achieved through an outgoing edge e with label l. If succ(v', l) returns null, this means e no longer exists in G', and then it is inserted in the set of deleted edges. Otherwise, if there is such a vertex, w', the algorithm checks whether w and w' are equivalent. If they are equivalent and w is not the exit vertex[3], the algorithm then checks whether w has a modification that do not alter the control flow, in which case the edge e is inserted in the set of retestable edges. Then the procedure Classify-edges is called recursively to continue the search.

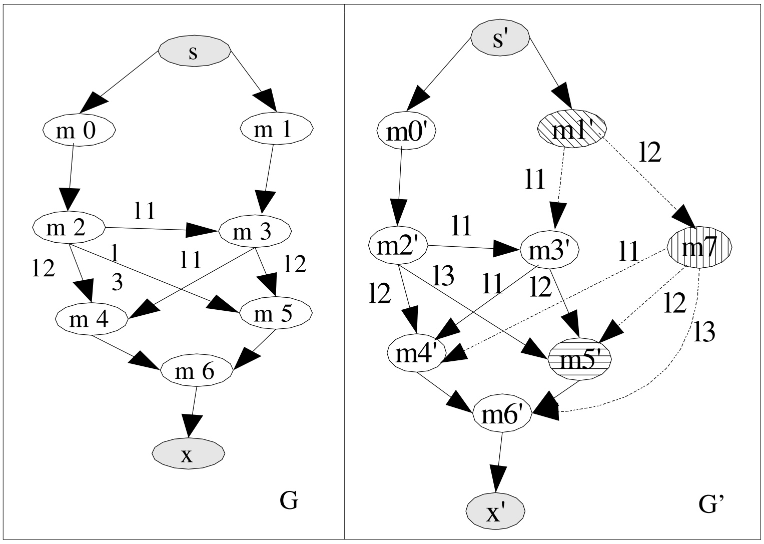


Fig. 3. Example graphs G and G'

To illustrate how the algorithm works, let us consider the two example graphs shown in Fig. 3.

In this example we suppose that m1' and m5' have the same signature as m1 and m5, respectively, but they are modifications of these two methods in the new version. In other words, V^c = {m1, m5}. The new vertex, m7, as well as the new edges, are represented as dotted lines in the figure. By applying the procedure Classify-edge to G and G', the results are the following sets:

D_e = {(m1, m3, ε)}

R_e = {(m3, m5, l2), (m2, m5, l3), (s, m1, l2)}

[3] The start vertices s and s' are considered as equivalent, as well as the exit vertices.

5.5 Test Case Selection

Now it is time to select $T' \subseteq T$, the reduced regression test suite. Using the sets D_e and R_e obtained in the previous step, we can classify the test cases in the following categories [7, 11, 15]:

- Reusable test cases (T_U): contains test cases whose paths p in $G(t)$ and p' in $G'(t)$ are equivalent and $p_v[i] \notin V^c$ for all i. In terms of the sets D_e and R_e, the set $T_U \subseteq T$ can be defined as follows: for a given test case t, if $D_e \cap E_p = \emptyset$ and $R_e \cap E_p = \emptyset$ then $t \in T_U$. These test cases are still valid and need not be rerun.
- Retestable test cases (T'): contains test cases whose paths p in $G(t)$ and p' in $G'(t)$ are equivalent and $\exists i \mid p_v[i] \in V^c$. These test cases are still valid and should be reapplied, maybe with some modifications, e.g., adjusting parameter values due to changes in a precondition. In terms of the D_e and R_e, the set $T' \subseteq T$ can be defined as follows: for a given test case t, if $D_e \cap E_p = \emptyset$ and $R_e \cap E_p \neq \emptyset$ then $t \in T'$.
- Obsolete test cases (T_O): a test case that traverses a path p in G that has no equivalent in G'. These test cases are no longer in accordance with the class specification and hence can no longer be reapplied. In other words, if $D_e \cap E_p \neq \emptyset$ then $t \in T_O$.

Test	Edges covered (E_p)
t1	{(s, m0, ε), (m0, m2, ε), (m2, m3, l1), (m3, m4, l1), (m4, m6, ε), (m6, x, ε)}
t2	{(s, m0, ε), (m0, m2, ε), (m2, m3, l1), (m3, m5, l2), (m5, m6, ε), (m6, x, ε)}
t3	{(s, m0, ε), (m0, m2, ε), (m2, m4, l2), (m4, m6, ε), (m6, x, ε)}
t4	{(s, m0, ε), (m0, m2, ε), (m2, m5, l3), (m5, m6, ε), (m6, x, ε)}
t5	{(s, m1, ε), (m1, m3, ε), (m3, m4, l1), (m4, m6, ε), (m6, x, ε)}
t6	{(s, m1, ε), (m1, m3, ε),(m3, m5, l2), (m5, m6, ε), (m6, x, ε)}

(a)

D_e = {(m1, m3, ε)}

R_e = {(m3, m5, l2), (m2, m5, l3), (s, m1, l2)}

T_O = {t5, t6}

T' = {t2, t4}

T_U = {t1, t3}

(b)

Fig. 4. Available test cases and regression-test-selection analysis results

As it can be deduced from the definitions above, the categories are pairwise mutually exclusive, i.e., $T_U \cap T' \cap T_O = \emptyset$.

To illustrate that, we use the example shown in Fig. 3. Fig. 4(a) shows the E_p sets for each test case of T, whereas Fig. 4(b) contains the defined sets.

5.6 Evaluation of the RTS Approach

RTS techniques can be compared on the basis of four criteria proposed by Rothermel and Harrold [20]. Their criteria are intended for code-based techniques, but with some adaptation they can also be used in our approach. The criteria are inclusiveness, precision, efficiency and generality.

Inclusiveness is the percentage of the test cases that may cause two versions of a program P and P' to behave differently. Inclusiveness is related to safety [1]: a safe RTS technique never eliminates a test t if the two versions behave differently on t. Under the assumptions in Section 5.4, our technique can be considered as inclusive, from which all test cases that exercise behavioral control flow modifications are selected. Besides, another assumption mentioned by Rothermel and Harrold, that the test suite should not contain obsolete test cases [21] is assured in our technique once obsolete test cases are identified and removed. However, implementation changes (c.f. Section 5.3) may not be "visible" to our technique, in which case certain test cases that exercise such modifications may not be selected. In that sense, the technique cannot be considered as safe. Nevertheless, there is a solution already pointed out in another research [6]: the expected implementation changes are informed to the test analyst in advance, so that they can be taken into account when constructing the V^c set, before code changes are implemented. With this information, all tests that exercise the modified method are selected, even some that could be eliminated, i.e, those which do not traverse the modified code. In this way, our RTS technique can be safe, but not precise. Precision is the percentage of selected test cases that do not produce different behavior in the new version.

Efficiency is relative to the space and time requirements of the RTS technique. Because test selection is based on a behavioral model rather than on source code, the analysis does not need the source code to obtain the structural control-flow graph or test coverage information. In addition to that, the analysis can be performed during the preliminary phase of regression testing (c.f. section 2), before any code change is implemented. However, an executable version of the behavioral model is necessary to perform the analysis.

Generality refers to the range of application of the RTS approach. Our technique applies when a behavioral model of the class exists, and it must be updated when the class changes. Although the testable class approach presumes a C++ implementation, the analysis is based on a behavioral model, so the technique is code-independent.

6 Case Studies

In this section we present the results of the application of our methodology on a case study, which is a real library developed by a research community, available at GNU site [10]. The CommonC++ is a C++ framework offering portable support for threading, sockets, file access, daemons, persistence, serial I/O and system services.

6.1 Subjects

Two classes of the CommonC++ library have been used as case studies: Socket and its derived class UDPSocket. The first one is an abstract class and its methods were only tested in the derived class context. The classes implement the datagram transmission protocol - UDP (User Datagram Protocol) through a socket implementation.

Table 1. Summary of the subjects used in the case study

			Public Interface		Test model (BCFG)	
Class	Version	LOC	Methods	Attributes	Vertices	Edges
Socket	1	668	16	0	15	34
	2	662	16	0	15	34
	3	703	16	0	15	34
	4	864	16	0	15	34
	5	864	16	0	15	34
	6	902	17	0	16	36
UDPSocket	1	127	8	0	23	49
	2	128	8	0	23	49
	3	152	8	0	23	49
	4	154	8	0	23	49
	5	154	8	0	23	49
	6	154	8	0	24	51

Table 1 gives a summary of the characteristics of the six versions of both classes used in this study. The methods were small, the shortest one having 2 LOC and the biggest one, 78 LOC.

6.2 Change Analysis

The differences between each pair of versions were obtained using the Unix utility, "diff". Table 2 shows the kind of information extracted from the comparison of classes Socket (C1) and UDPSocket (C2). The modifications were classified according to the categories presented in Section 5.3. The table shows the total number of methods and attributes changed (added, changed or deleted). The last two rows present the total number of contract and implementation changes. As it can be noticed, the second version contains only contract modifications. In fact, the modifications consisted of changing the method's signature, which in our analysis corresponds to a removal of an old method and an addition of a new one.

Table 2. Impact Analysis Results

	Total v.1		Added		Changed		Deleted		Total v.2	
	C1	C2	C1	C2	C1	C2	C1	C2	C1	C2
Attributes	5	1	0	0	1	0	0	0	5	1
Methods	16	8	3	4	4	0	3	4	16	8
Contract	16	8	0	0	4	0	0	0	16	8
Implementation	0	0	0	0	4	4	0	0	0	0

6.3 Test Case Selection

The bar charts in Fig. 5 and 6 summarize the results of the RTS analysis for the two classes. The Y-axis shows the amount of test cases. The X-axis contains for each version: the size of the test suite (including new test cases), as well as the number of reusable (RE), retestable (RT) and obsolete (O) test cases.

As it can be observed, the size of the test suite did not vary from one version to another – it remained constant for the Socket class. From the regression testing point of view, version 2 was the worst because the number of obsolete test cases was the highest (68% for UDPSocket). However, this was expected according to the report shown in Table 2.

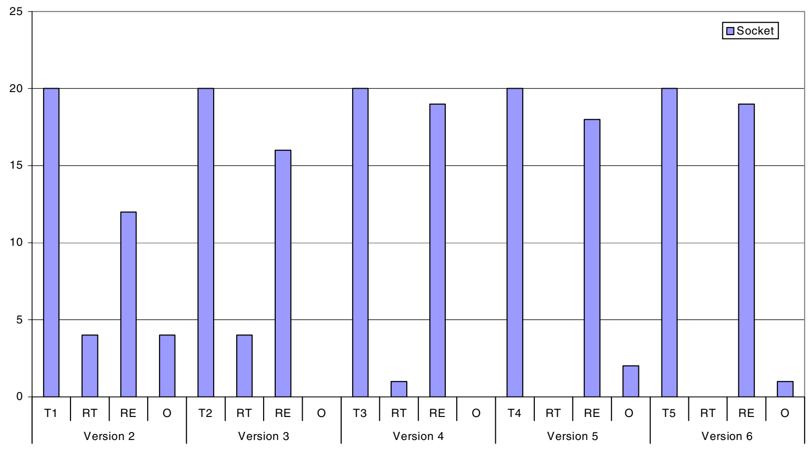


Fig. 5. The graph of the test suite for each version of class Socket

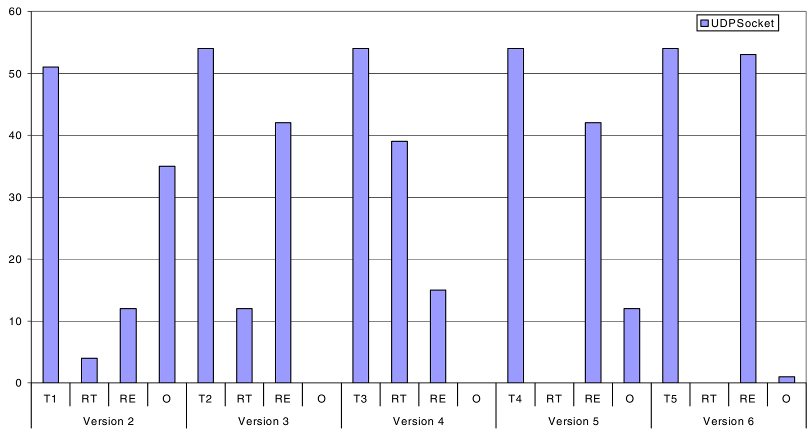


Fig. 6. The graph of the test suite for each version of class UDPSocket

It is also worth noting that for all versions except the second, the number of reusable test cases is high, meaning that most modifications did not affect the class behavior. Moreover, for versions 5 and 6 no tests needed to be reapplied, as the change consisted of the addition of a new method to the interface of the class Socket. New test cases were generated to test this new method though.

Finally, to determine whether the selected test cases really traversed a modification in the code, we inserted, at each modified location, a statement that writes "`>>> method name: modification_id`" to a log file. The result was that all the modifications were traversed by the retestable test cases. We repeated the same experiments with the reusable test cases and the result was that no modifications were traversed.

Results are promising, in which savings in retesting efforts were achieved without compromising the safety of the technique. Of course, further experiments are necessary with more diverse subjects to generalize the results obtained here.

7 Conclusions and Future Works

This paper has presented an approach for selecting regression tests for testable classes. The technique is specification-based, in which a behavior model of a class is considered. This model represents the different ways to create an object, the different tasks it starts and the different ways to destroy it. The technique selects test cases from existing test suites that traverse the different paths in the class behavior model. The approach adapts an algorithm already proposed in the literature by Rothermel and Harrold, to use a behavioral instead of a source code-based model. The technique is not safe as theirs in selecting test cases that effectively exercise the modifications, unless implementation changes are taken into account. However, since it is not source-code dependent, it presents the following advantages: (i) it can be applied in the preliminary phase of regression testing, when the implementation of the changes is not yet complete. The test and the development analysts can then work in parallel, which saves time; (ii) it can be applied by users who may not have access to the class source code.

In addition to that, the regression testing activity can also benefit from the use of a testable class because, besides the behavioral model, the class also contains built-in testing capabilities that are helpful during testing. One of these capabilities is the implementation of the class contract that can help in the results' analysis.

Although the current work addressed C++ software, it can be applied to classes written in other languages, once the selection of test cases is not source code dependent.

An empirical study using a real world class library was performed to show the applicability of the technique. Of course, further experimentations are being envisaged. In the short term, a comparison with other techniques, even white-box ones, is being considered.

In the long term, we envisage the integration of our RTS technique to a test case generation tool developed in a previous work. In this way, users can have a unique tool to develop new test cases as well as to select tests from an existing test suite.

Additional future works consider loops and concurrence. Besides, the proposed technique considers a class and its subclasses, but does not consider the interaction among classes. This will also be the subject of another study.

Acknowledgments

We thank the anonymous reviewers that provided many comments and suggestions and whose thorough review and constructive comments greatly contributed to the improvement of this paper.

References

1. Ball, T.: On the Limit of Control-Flow Analysis for Regression Test Selection. In International Symposium on Software Testing and Analysis (ISSTA), 1998, pp143-242.
2. Beizer, B.: Black-Box Testing. John Wiley & Sons, 1995.
3. Beydeda, S.; Gruhn, V.: Merging components and testing tools – The self-testing COTS components (STECC) strategy. In: Proc. EUROMICRO Conference Component-based Software Engineering Track, 2003.
4. Binder, R.V.: Design for Testability in Object-Oriented Systems. Comm. of ACM, 37(9), Sept./1994, pp87-101.
5. Binder, R.V.: Testing Object-Oriented Systems. Models, Patterns and Tools. Addison-Wesley, 2000.
6. Briand, L.; Labiche, Y.; Soccar, G.: Automating Impact Analysis and Regression Test Selection Based on UML Designs. International Conference on Software Maintenance (ICSM'02), p.0252, Montreal, Quebec, Canada, October/2002.
7. Chen, Y.; Probert, R.L.; Sims, D.P.: Specification-based regression test selection with risk analysis. Conference of the Center for Advanced Studies on Collaborative Research, pp1-14, 2002.
8. Fowler, M.; Scott, K.: UML Distilled. Second Edition, Addison-Wesley, 2000.
9. Gao, J.: Testing Component-Based Software. Proc. of STARWest, San Jose, USA, 1999. Available on the Web: www.engr.sjsu/edu/gaojerry/report/star-test.htm.
10. GNU – CommonC++: ftp://ftp.gnu.org/pub/gnu/commonc++/ Accessed in January/2002.
11. Granja, I.; Jino, M.: Techniques for Regression Testing: Selecting Test Case Sets Tailored to Possibly Modified Functionalities. Third IEEE European Conference on Software Maintenance Engineering, March/1999, pp.2-11.
12. Harrold, M.J; Jones, J.A.; Li, T.; Liang, D.; Orso, A.; Pennings, M.; Sinha, S.; Spoon, S.A.: Regression Test Selection for Java Software. In ACM Conf. on OO Programming, Systems, Languages and Applications (OOPSLA), 2001.
13. Hoffman, D.: Hardware Testing and Software ICs. Proc. Northwest Software Quality Conference, USA, Sept/1989, pp234-244.
14. Hörnstein, J., Edler, H.: Configuration and testing of components in software product lines, Proc. SERPS'02: Second conference on software engineering research and practise in Sweden, Karlskrona, Sweden, 24-25 October 2002.
15. Leung, H.K.N; White, L.: Insights into Regression Testing. Proc. Of the IEEE Conference on Software Maintenance, Ocotober/1989, pp531-537.
16. Leung, H.K.N; White, L.: A Cost Model to Compare Regression Test Strategies. Proc. Of the IEEE Conference on Software Maintenance, Ocotober/1991, pp201-208.
17. Martins, E.; Toyota, C.; Yanagawa, R.: Constructing Self-Testable Software Components. In: Proc. IEEE/IFIP Dependable Systems and Networks (DSN) Conference. Gothemburg, 2001.
18. Meyer, B.: Applying Design by Contract. IEEE Computer, Oct./1992, pp40-5.

19. Orso, A.; Harrold, M.J.; Rosenblum, D.: Component Metadata for Software Engineering Tasks. Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000), LNCS Vol. 1999, Springer November 2000, Davis, CA, USA, pp. 129-144.
20. Rothermel, G.; Harrold, M.J.: Selecting regression tests for object-oriented software. IEEE Procedings of International Conference on Software Maintenance, Sep/1994, pp14-25.
21. Rothermel, G.; Harrold, M.J.: Analyzing Regression Test Selection Techniques. IEEE Transactions on Sw. Engineering, 22(8), 1996.
22. Rothermel, G.; Harrold, M.J.: A Safe, Efficient Regression Test Selection Technique. ACM Transactions on Software Engineering and Methodology, 1997, pp173-210.
23. Rothermel, G.; Harrold, M.J.; Dedhia, J.: Regression Test Selection for C++ Software. Journal of Software Testing, Verification and Reliability,10(2), June/2000.
24. Traon, Y.; Deveaux, D.; Jézéquel, J.: Self-testable Components – from Pragmatic Tests to Design-for-Testability Methodology. In: Proc. 1st. Workshop on Component-Based Systems, Switzerland, 1997.

Author Index